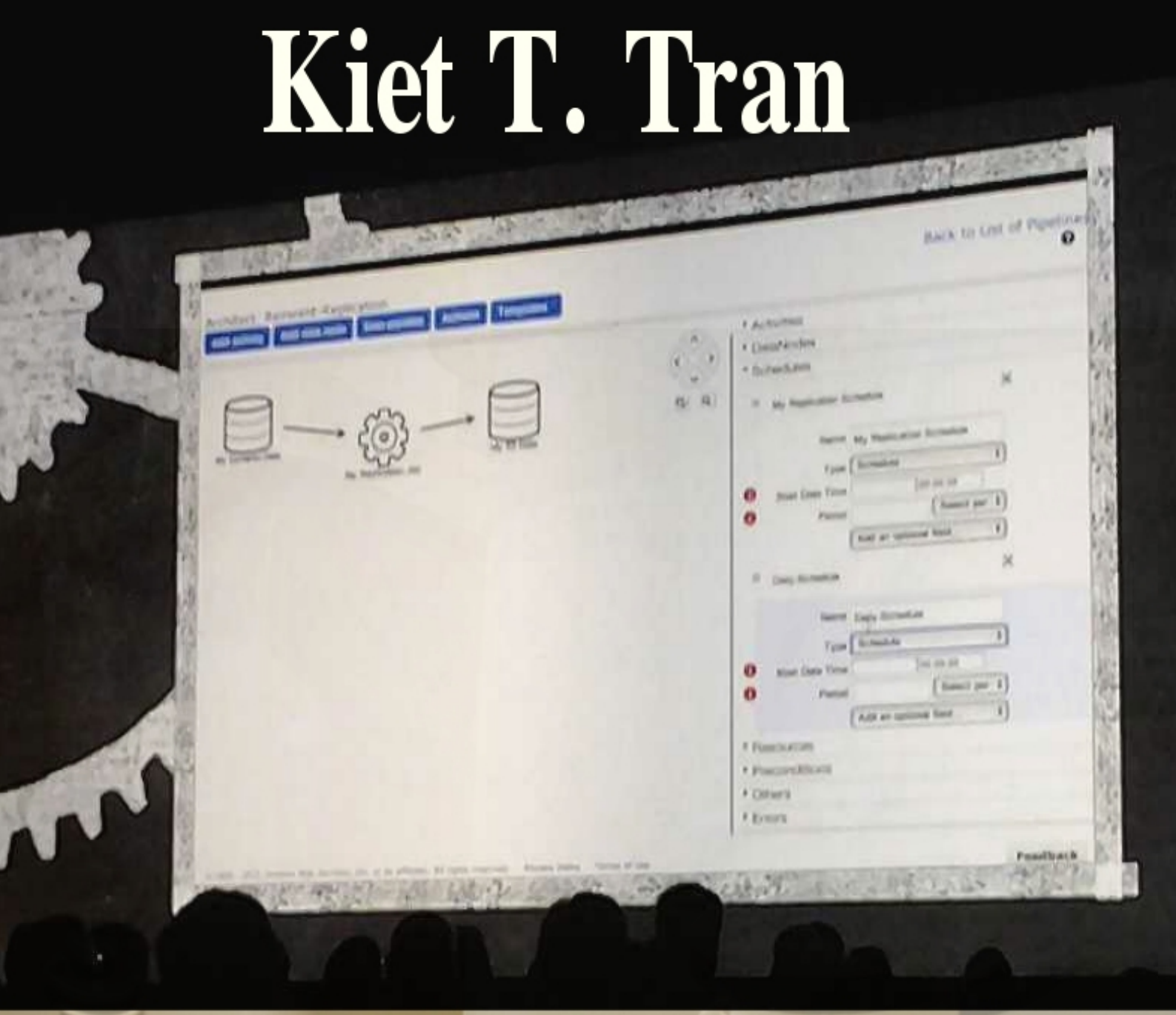
Kiet T. Tran



Introduction to Web Services with Java

Kiet T. Tran, PhD

Introduction to Web Services with Java

Introduction to Web Services with Java
1st edition

ISBN 978-87-403-0509-8

Contents

Preface

This book, which is based on lectures I provided at Trident University International, focuses more on the details of a hands-on approach to Web Service programming than its specifications; however, in order to help readers grasp the concept more easily, we provide a brief introduction to Web Service, SOAP, and WSDL in the first three chapters. Many details of the specifications are intentionally glossed over, however, so that the content remains manageable.

Objectives

- To understand the basic concepts of Web Services:
  - SOAP
  - WSDL
- To develop basic Web Services using the following major programming tools:
  - Java JDK 6 or later
  - Apache CXF 2.7 or later
  - Oracle Middleware WebLogic Server (WLS) 12.1 or later

Background requirements for this book:

- Basic Java programming experience
- Basic understanding of Web programming
- Basic understanding of XML

Web Service (WS) is a technology, process, and software paradigm that provides support for business integrations mainly over an Internet-based environment. This book presents basic concepts of WS, protocol stack, and applications. In addition to studying the three supporting standards SOAP, WSDL, and UDDI, students will learn how to implement WS using Java-centric technologies such as JAXP, JAXRPC, SAAJ, and JAXB. Students will also study how business processes can be implemented using WS via BPEL.

WS is a software application identified by a URI whose interfaces and binding are capable of being defined, described, and discovered by XML artifacts, and it supports direct interactions with other software applications using XML-based messages via Internet-based protocols (W3C: http:// www.w3.org/TR/ws-desc-reqs). WS is mainly for machine-to-machine communication. The WS standard relies on other standards – namely, SOAP, WSDL, and UDDI – to function efficiently. SOAP is an application protocol that is used to transmit messages between a WS client and a WS server. HTTP is the transport protocol of choice for SOAP; however, JMS and SMTP protocols have also been used. WSDL is used to describe the service that an external application can call. UDDI is used to publish and advertise services so that they can found and used by others. UDDI also uses SOAP as its application protocol.

- Chapter 1 – Introduction. This chapter provides an overview of Web Services. It presents a brief background of past enterprise integration difficulties and the benefits that a WS can offer. This chapter introduces a beginning-level WS program using Java Web Services.
- Chapter 2 – SOAP. Using the example from the previous chapter (i.e., a WS that exchanges a complex data structure), this chapter explains how basic SOAP message exchange works.
- Chapter 3 – WSDL. This chapter offers an in-depth look at how a service can be described and understood by others. A more complex WS application is then built upon the previous module exercise.
- Chapter 4 – A Sample Application
- Chapter 5 – Apache CXF and Tomcat Server
- Chapter 6 – Deployment of CXF Application on Oracle WebLogic Server 12
- Chapter 7 – Appendix A – Development Environment

The book focuses on the working mechanism of WS with a hands-on programming exercise using a basic Java WS framework. This framework works on a standalone Java application, an Oracle WebLogic Server (WLS), and an Apache Tomcat server. Thus, readers are expected to have sufficient knowledge of Java and XML.

List of Figures

Table of Listings

Table of Tables

1 Introduction

Objectives

After studying this chapter, you should be able to:

1. Describe basic elements of a Web Service application
2. Compare and contrast the purposes of Web and Web Service applications
3. Describe the benefits of Web Services
4. Write a simple Web Service application using Java Development Kit (JDK) 6 or later
5. Verify and test a Web Service application

In the early days of the Internet, Web applications delivered static webpages via HTML. Certainly, the development of websites was simpler; however, static content can quickly become outdated; thus, the content management of a website is important.

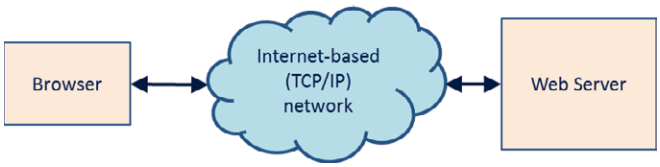


Figure 1-1 *Early Web applications*

In order to provide dynamic content to Web users, 2-tier web applications were realized with the introduction of the Common Gateway Interface (CGI), which retrieves content from external data resources, such as a database. CGI acts as a client in the traditional client-server architecture. A CGI script processes the request and returns the result to the Web server. The server then formats the contents in HTML and returns to the browser for display.

CGI suffered many drawbacks that necessitated changes to the 2-tier architecture. The database was often running on the same machine; therefore, making backups of the data was difficult. CGI was running as a separate process, so it suffered from a context-switching penalty. CGI was not designed for performance, security or scalability.

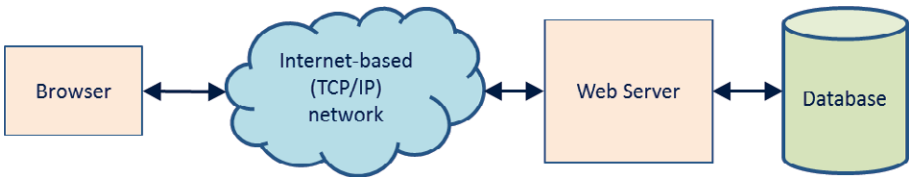


Figure 1-2 *Two-tier web application*

Nowadays, n-tier Web application architecture is commonly used. In this architecture, middleware or an application server is introduced to connect the Web server and the database more efficiently. The performance of an n-tier application is improved because Web servers, middleware and databases can be hosted by separate machines. Each tier can be replicated for the purposes of load balancing. Security is also improved because data is not stored on the Web or application server, which makes it harder for hackers to gain access into the database where data is stored.

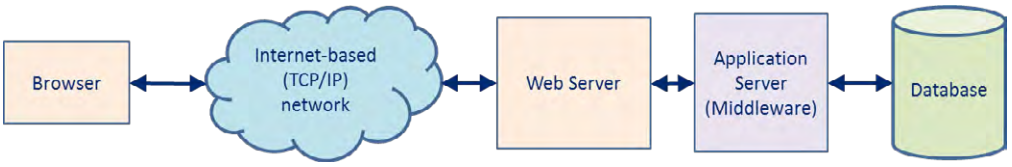


Figure 1-3 *An n-tier web architecture*

A web and an application servers are often run on the same machine; however, it is best practice to run the database server on a separate machine. In a software development environment, all three servers can be hosted on a single machine. In this book, a server is often refered to a software application.

1.1 Browsing the Internet

Before the conception of Web 2.0 (around 1999), the basic use of the World Wide Web (WWW) and the Internet was simple and based on the traditional client-server model with older technologies such as Remote Procedure Call (RPC) or Transaction Processing (TP) Monitors or other middleware that permitted programmable clients.

Consider a typical use case of a person browsing the Internet by means of a browser. The Web server in this example serves dynamic HTML pages using Java Server Pages (JSP) technology. In addition, it uses Enterprise Java Bean (EJB) or Plain-Old-Java-Object (POJO). JSP is oriented toward the delivery of webpages for the presentation layer. EJBs or POJOs are usually used for processing business rules. There are thousands of Web applications that use Java/JEE technology.

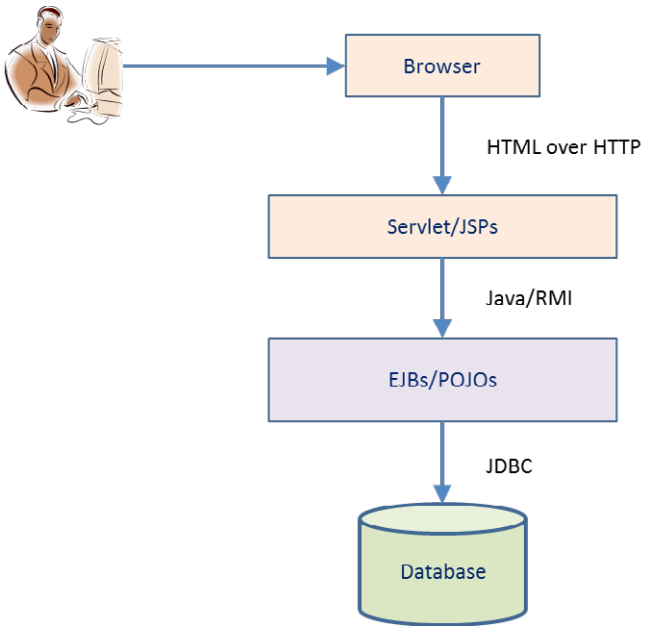


Figure 1-4 *Man-machine interaction*

The Internet architecture was originally designed for human users. HTTP protocol was for exchanging documents (Web or HTML pages). HTML was designed for basic graphical user interface (GUI) applications. Computing resources on a web browser are often idle while the user is browsing the Internet. These available resources prompted the idea of providing more robust web browsing experience. In addition, the idea of business-to-business (B2B) data exchange model also became more feasible. Accordingly, the WS architecture was introduced to support this new type of data exchange.

1.2 Web Service architecture

A service can be one of the three types of interaction: man-to-man, man-to-machine, or machine-to-machine. A restaurant service is an example of man-to-man interaction. A person withdrawing money from an Automated Teller Machine (ATM) is an example of man-to-machine interaction. Machine-to-machine interaction is exemplified by a handheld device, such as a smart device (e.g., a phone or a tablet), synchronizing its address book with Microsoft Outlook. A Web Service is a type of machine-to-machine interaction that uses specific Web standards and technology. A Web Service is a set of programming interfaces, not a set of webpages.

This section begins with a basic definition of a Web Service in order to establish a basic understanding for use in later chapters. More complex aspects of Web Services will be easier to understand when the basic concept of a Web Service is properly explained.

According to W3C website, http://www.w3.org/TR/ws-desc-reqs:

> *A Web Service is a software application identified by a URI whose interfaces and binding are capable of being defined, described and discovered by XML artifacts and [that] supports direct interactions with other software applications using XML based messages via Internet-based protocols.*

A Web Service must involve a Web-based protocol, such as HTTP or Simple Mail Transfer Protocol (SMTP). Other transport protocols may be used, but HTTP is the most common one being used. HTTPS uses Secure Socket Layer (SSL) or Transport Secure Layer (TLS) for secured transport of data. In regard to software development concerns, the difference between HTTP and HTTPS is trivial. HTTP, thus, is used throughout this text.

A Web Service is a software application that requires interaction with another application. WS is a software integration technique for a B2B type of integration. Here, one application acts as a service provider (server) and the others act as service consumers (clients). This is a many-to-one relationship.

'Interface' is defined as "[The] point of interaction or communication between a computer and any other entity" (http://www.thefreedictionary.com). An interface can also be described as an "abstraction of a service that only defines the operations supported by the service (publicly accessible variable, procedures, or methods), but not their implementation" (Szyperski, 2002). For example, in Java, an interface can be defined and then implemented by a concrete class.

Web Service Description Language (WSDL) specifies the service interface and the rules for binding the service consumer and the provider. According to the specification of WSDL 1.1, WSDL is defined as "an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information" (http://www.w3.org/TR/wsdl). WSDL defines how a consumer can interact with a service via a concrete network protocol and message format using eXtended Markup Language (XML).

XML is a profile (subset) of Standard Generalized Markup Language (SGML). SGML is a metalanguage, i.e., a language that describes other languages. Unlike HyperText Markup Language (HTML), which is used to serve static webpages, XML allows the author to create his or her own tags. Thus, XML facilitates the data and document processing functions.

Web Service relies on Simple Object Application Protocol (SOAP) as its transport. As its name implies, SOAP is a lightweight protocol that can be used to exchange structured messages (i.e., XML). SOAP 1.2 is the latest version. WSDL 1.1 supports SOAP 1.1, HTTP GET/POST, and MIME.

A service can be defined, published and discovered using some type of service registry. Current supporting service registries include electronic business XML (ebXML), Universal Discovery, Description and Integration (UDDI), and Metadata Registry (MDR). UDDI is usually a good idea; however, it is not widely used except in a private network of services.

RPC is a powerful technique that provides distributed computing capabilities across a network of machines. RPC is a form of interprocess communication that enables function calls between applications that are located across different (or the same) locations over a network. It is best suited for client-server programming.

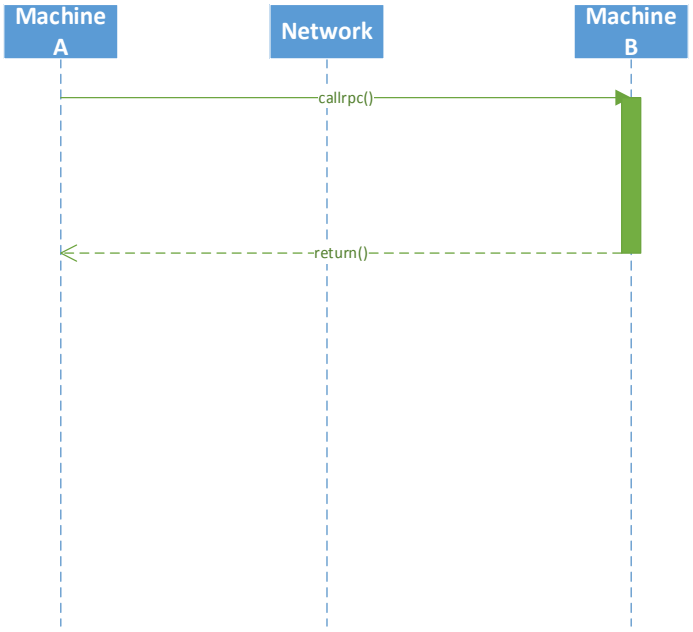


Figure 1-5 *Remote Procedure Call (RPC)*

Web Services can be used to help solve several problems in Enterprise Application Integration (EAI). Integrating existing applications for a business solution is a complex and time-consuming task. Applications that were written in different computer languages, such as C/C++, JAVA, Visual Basic, and FORTRAN, have unique logical interfaces to the external world, which makes the integration of these applications difficult, complex and time-consuming. Applications that are running on different machine architectures, such as SUN, Personal Computer, IBM Mainframes, IBM A/S 400, have unique physical interfaces to the external world. Integrating these applications is also challenging. Applications running on machines that are interconnected through a network are also difficult to integrate. The challenges of EAI arise in three main areas:

- Language barriers – XML is a standardized language that is used for message exchange
- Platform barriers – SOAP has been implemented on many platforms (e.g., Unix, Windows)
- Network barriers – HTTP and SMTP are standardized network protocols

WS can serve as an enabling technology for application integration. WS, as mentioned earlier, places thes following major standards in focus: XML, SOAP, WSDL, UDDI and HTTP.

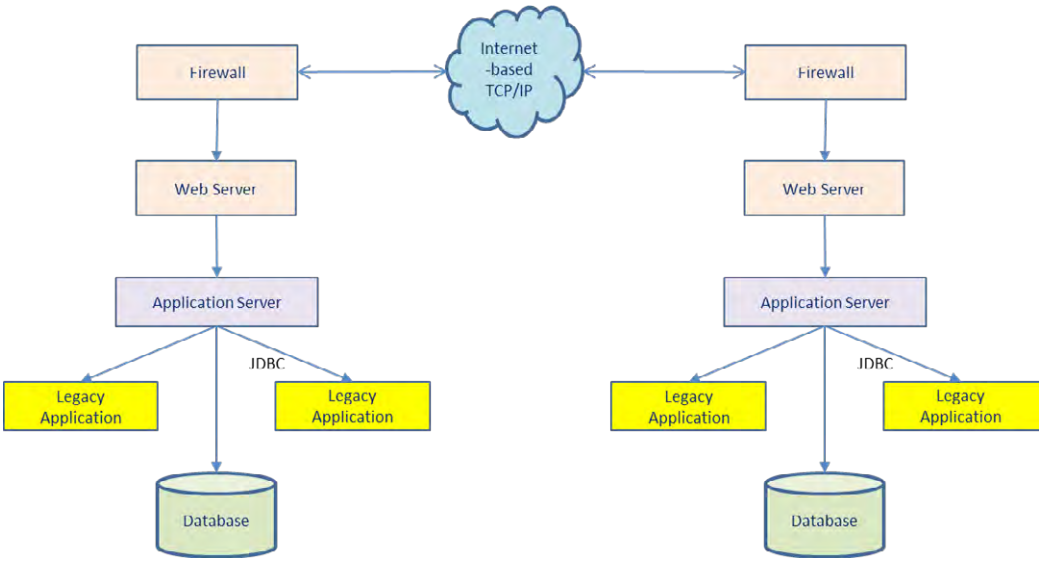


Figure 1-6 *Business-to-Business integration*

SOAP sequence diagram

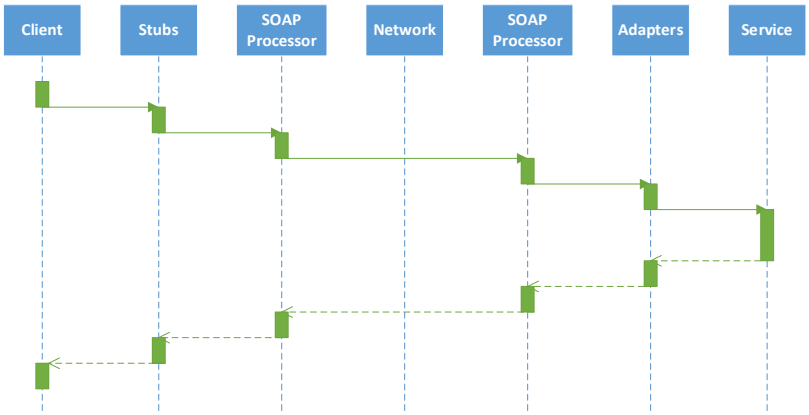


Figure 1-7 *Sequence diagram of SOAP*

Service requester – the client that consumes or requests the service
Service provider – the entity that implements the service and fulfill the service requests
Service registry – a listing like a phonebook where available services are listed and described in full

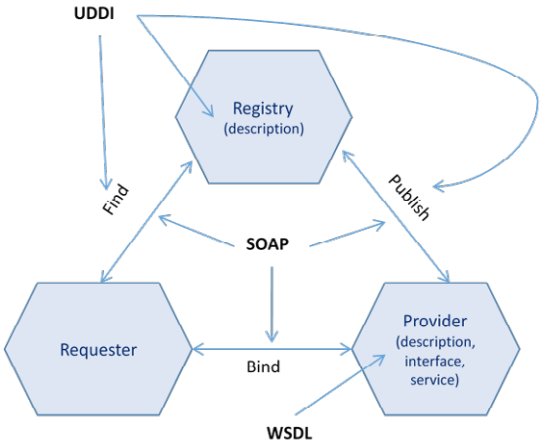


Figure 1-8 *Web Service Architecture*

1.3 Benefits of Web Services

Web Services provide many benefits:

1. Platform-independent: Web Services are now available in nearly all platforms:
   a) Hardware: mainframe, midrange, personal and mobile devices
   b) Operating systems: UNIX, Windows, Mainframe OS, Android, and iPhones
2. Reuse of existing networking infrastructure: HTTP, SMTP, and JMS protocols
3. Loose-coupling of software components promotes software reuse
4. Reduced integration cost and increased integration speed
5. Open architecture and communication protocols

1.4 Program a HelloWorld Web Service

The concept of a Web Service can be difficult to comprehend without seeing a concrete example of how a Web Service is created and used. The top-down approach starts with a WSDL file that describes the services. The top-down approach may increase the level of interoperability and allow more control of the WS, wehereas the bottom-up approach starts at the low level of the Java bean or enterprise Java bean (EJB) and is faster and easier.

The following steps can be used to create and test a simple WS application:

1. Run Eclipse IDE, create a new Java project, and name it 'java-ws'.
2. Run Server.java as a Java program.
3. Verify the WSDL and the associated schema for the service endpoint: http://localhost:9999/HelloWorld?wsdl.
4. Use SOAPUI software to test the HelloWorld Web Service.
5. Create Java Web Service client code.

1.4.1 Create a Project

In the example above, the bottom-up approach is used. This example requires Java 6 or later. A Web Service called 'Hello World' is created with the method called 'say', which requires one String parameter. To create a Java project under Eclipse IDE, perform the following steps:

1. Run Eclipse IDE.
2. Choose File → New → Java Project. Use all default and name it 'java-ws'.
3. Expand the java-ws project, then right-click on the src directory and choose New → Package. Name the package 'com.bemach.ws.hello'.
4. Similarly, create another Java package com.bemach.ws.server.
5. Create two Java classes – com.bemach.ws.hello.HelloWorld.java and com.bemach.ws.server.Server.java.

1.4.2 Create a Web Service

A classic HelloWorld class of Java can be written in a few lines of code. The purpose is to make sure that a Java Virtual Machine is properly installed and ready for programming.

Listing 1-1. HelloWorld.java

```
package com.bemach.ws.hello;
/**
 * 2013 (C) BEM, Inc., Fairfax, Virginia
 *
 * Unless required by applicable law or agreed to in writing,
 * software distributed is distributed on an
 * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
 * KIND, either express or implied.
 *
 */

public class HelloWorld {
   public String say (String name) {
      return String.format("Hello, %s!", name);
   }

   public static void main (String[] args) {
      String msg = new HelloWorld().say("Johnny, B. Good");
      System.out.println(msg);
   }
}
```

Output:

```
Hello, Johnny B. Good!
```

In this way, the HelloWorld program is transformed into a WS application. This is a basic WS application using the reference implementation of JAX-WS by the Java language.

To transform the HelloWorld program from a simple Java bean into a Java WS application, four WS annotations – namely, @WebService, @SOAPBinding, @WebMethod and @WebParam – are decorated as follows:

Listing 1-2. HelloWorld.java with Web Service Annotations

```
package com.bemach.ws.hello;
/**
 * 2013 (C) BEM, Inc., Fairfax, Virginia
 *
 * Unless required by applicable law or agreed to in writing,
 * software distributed is distributed on an
 * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
 * KIND, either express or implied.
 *
 */

import java.util.logging.Logger;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;

@WebService
@SOAPBinding(style=SOAPBinding.Style.DOCUMENT)
public class HelloWorld {
   private static final Logger LOG = Logger.getLogger(HelloWorld.class.getName());

   @WebMethod
   public String say (@WebParam(name="name") String name) {
      LOG.info("Web service is called!");
      return String.format("Hello, %s!", name);
   }

   public static void main (String[] args) {
      String msg = new HelloWorld().say("Johnny, B. Good");
      LOG.info(msg);
   }
}
```

Annotations indirectly affect the sematics of the program via tools and libraries. The @WebService annotation indicates that the class will implement a WS. The @SOAPBinding annotation indicates the style of the SOAP to be used. In this example, the style is DOCUMENT as opposed to RPC. @WebMethod indicates an operation of the WS to be created. Lastly, the @WebParam indicates how the parameter is named inside the WSDL.

1.4.3 Create a HTTP Server

To host the service endpoint, a WS requires an HTTP server. An Apache JEE Tomcat server can be used; however, a basic server can be created using only Java. In this example, we created an Endpoint with a specific URL that ties to an implementation of the WS. In this case, the URL is http://localhost:9999/java-ws/hello and the implementation is the HelloWorld object svc.

Listing 1-3. Server.java class

```
package com.bemach.ws.server;

/**
 * 2013 (C) BEM, Inc., Fairfax, Virginia
 *
 * Unless required by applicable law or agreed to in writing,
 * software distributed is distributed on an
 * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
 * KIND, either express or implied.
 *
 */

import java.util.logging.Logger;

import javax.xml.ws.Endpoint;
import javax.xml.ws.EndpointReference;

import com.bemach.data.DbConfig;
import com.bemach.ws.doc.employees.EmployeeDocData;
import com.bemach.ws.hello.HelloWorld;
import com.bemach.ws.rpc.employees.EmployeeRpcData;

/**
 *
 */
public final class Server {
   private static final Logger LOG = Logger.getLogger(Server.class.getName());
   private static final String MYSQL_DRIVER="com.mysql.jdbc.Driver";
   private static final String DB_HOST = "saintmonica";
   private static final String DB_PORT = "3306";
   private static final String DB_SID = "employees";
   private static final String DB_USER = "empl_1";
   private static final String DB_PSW = "password";
   private Server() {

   }

   protected static DbConfig getDbConfig() {
      DbConfig dbCfg = new DbConfig();
      dbCfg.setDriverName(MYSQL_DRIVER);
      dbCfg.setHost(DB_HOST);
      dbCfg.setPort(DB_PORT);
      dbCfg.setDb(DB_SID);
      dbCfg.setUid(DB_USER);
      dbCfg.setPsw(DB_PSW);
      return dbCfg;
   }

   private static final String HOST_NAME = "localhost";
   private static final String PORT_NO = "9999";
   private static final String HELLO_SVC_NAME = "java-ws/hello";
   private static final String RPC_EMPL_SVC_NAME = "rpc/employees";
   private static final String DOC_EMPL_SVC_NAME = "doc/employees";
   private static final String PROTOCOL = "http";
```

```
protected static SvrConfig getSvrConfig() {
   SvrConfig svrCfg = new SvrConfig();
   svrCfg.setListenHostname(HOST_NAME);
   svrCfg.setListenPort(PORT_NO);
   svrCfg.setListenInterface(HELLO_SVC_NAME);
   svrCfg.setListenProtocol(PROTOCOL);
   return svrCfg;
}

protected static Endpoint publish(SvrConfig cfg, Object svc) {
   String url = String.format("%s://%s:%s/%s",
            cfg.getListenProtocol(),
            cfg.getListenHostname(),
            cfg.getListenPort(),
            cfg.getListenInterface());
      Endpoint ep = Endpoint.publish(url, svc);
      EndpointReference epr = ep.getEndpointReference();
      LOG.info("\nEndpoint Ref:\n"+epr.toString());
      return ep;
}

protected static void startHelloWorld() {
   SvrConfig cfg = getSvrConfig();
   cfg.setListenHostname(HOST_NAME);
   cfg.setListenInterface(HELLO_SVC_NAME);
   cfg.setListenPort(PORT_NO);
   cfg.setListenProtocol(PROTOCOL);

   HelloWorld hello = new HelloWorld();
      publish(cfg, hello);
      LOG.info("HelloWorld service started successfully ...");
}

protected static void startRpcEmployees() {
   SvrConfig svrCfg = getSvrConfig();
   svrCfg.setListenHostname(HOST_NAME);
   svrCfg.setListenInterface(RPC_EMPL_SVC_NAME);
   svrCfg.setListenPort(PORT_NO);
   svrCfg.setListenProtocol(PROTOCOL);
      DbConfig dbCfg = getDbConfig();
      svrCfg.setDbCfg(dbCfg);

   EmployeeRpcData rpcEmpl = new EmployeeRpcData(dbCfg);
      publish(svrCfg, rpcEmpl);
      LOG.info("Employees (RPC) service started successfully ...");

}
```

```
   protected static void startDocEmployees() {
      SvrConfig svrCfg = getSvrConfig();
      svrCfg.setListenHostname(HOST_NAME);
      svrCfg.setListenInterface(DOC_EMPL_SVC_NAME);
      svrCfg.setListenPort(PORT_NO);
      svrCfg.setListenProtocol(PROTOCOL);
         DbConfig dbCfg = getDbConfig();
         svrCfg.setDbCfg(dbCfg);

      EmployeeDocData docEmpl = new EmployeeDocData(dbCfg);
         publish(svrCfg, docEmpl);

         LOG.info("Employees (Document) service started successfully ...");
   }

   /**
    * Start WS Server with multiple service endpoints...
    *
    * @param args
    */
   public static void main(String[] args) {
      startHelloWorld();
      startRpcEmployees();
      startDocEmployees();
 }
}
```

When the Server program runs, it calls the startHelloWorld method to create a WS implementation that ties with a unique URL. The endpoint is then published and ready for receiving requests from a remote client. For this simple program, Ctrl-C can be used to stop the server.

The DbConfig.java class is a simple placeholder for the required parameters for database access and for the URL. In a later example, Java code is used to implement data access to the database. Remember, though, that this is sample code; therefore, the password is displayed or stored in the clear. In a business or secure environment, passwords are entered each time or stored encrypted.

1.5 Host a Web Service

In Eclipse IDE, perform the following actions:

1. To run, open Server.java class. Choose Run → Run As → Java Application.

An Eclipse project would look like this:

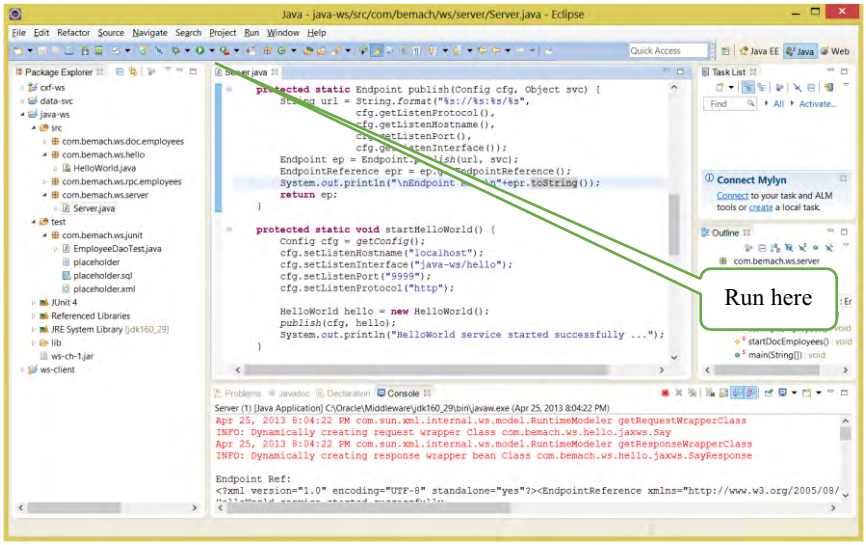


Figure 1-9 *An Eclipse Java project for the HelloWorld Web Service*

1.6 Verify a Web Service

View the HelloWorld's WSDL:

Open a browser and go to this URL: http://localhost:9999/java-ws/hello?WSDL

Listing 1-4. HelloWorld WSDL

```
<definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://hello.ws.bemach.com/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
targetNamespace="http://hello.ws.bemach.com/"
   name="HelloWorldService">
   <types>
      <xsd:schema>
         <xsd:import namespace="http://hello.ws.bemach.com/"
            schemaLocation="http://localhost:9999/java-ws/hello?xsd=1" />
      </xsd:schema>
   </types>
   <message name="say">
      <part name="parameters" element="tns:say" />
   </message>
   <message name="sayResponse">
      <part name="parameters" element="tns:sayResponse" />
   </message>
   <portType name="HelloWorld">
      <operation name="say">
         <input message="tns:say" />
         <output message="tns:sayResponse" />
      </operation>
   </portType>
   <binding name="HelloWorldPortBinding" type="tns:HelloWorld">
      <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
         style="document" />
      <operation name="say">
         <soap:operation soapAction="" />
         <input>
            <soap:body use="literal" />
         </input>
         <output>
            <soap:body use="literal" />
         </output>
      </operation>
   </binding>
   <service name="HelloWorldService">
      <port name="HelloWorldPort" binding="tns:HelloWorldPortBinding">
         <soap:address location="http://localhost:9999/java-ws/hello" />
      </port>
   </service>
</definitions>
```

Encoding style

Service endpoint

Figure 1-10. *The WSDL of the HelloWorld Web Service*

To view the associated XML schema, go to this URL: http://localhost:9999/java-ws/hello?xsd=1

Listing 1-5. HelloWorld XSD

```
<xs:schema xmlns:tns="http://hello.ws.bemach.com/"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
   version="1.0" targetNamespace="http://hello.ws.bemach.com/">
   <xs:element name="say" type="tns:say" />
   <xs:element name="sayResponse" type="tns:sayResponse" />
   <xs:complexType name="say">
      <xs:sequence>
         <xs:element name="name" type="xs:string" minOccurs="0" />
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="sayResponse">
      <xs:sequence>
         <xs:element name="return" type="xs:string" minOccurs="0" />
      </xs:sequence>
   </xs:complexType>
</xs:schema>
```

Figure 1-11. *The XML schema associated with the HelloWorld Web Service.*

1.7 Test a Web Service with SOAPUI

SOAPUI is a software that enables software developers and integrators to test Web Services. Similar to Eclipse IDE, SOAPUI is a project-based application.

1. Run SOAPUI program.
2. Select File → New SOAPUI project.
3. Fill in the Project Name and the Initial WSDL/WADL.

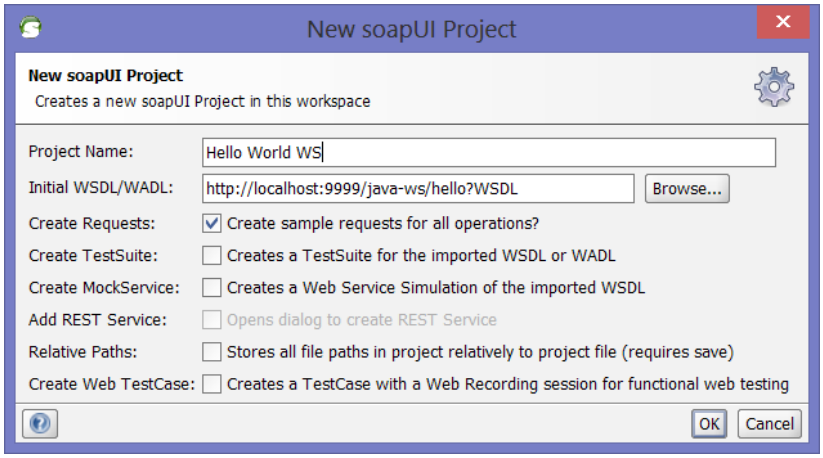


Figure 1-12 *Create a SOAPUI project for the HelloWorld Web Service*

To execute a SOAP operation, take the following steps:

1. On the left panel, double-click on Request 1.
2. Fill in the blank between <arg0> and </arg0>.
3. Click on the green triangle on the top left panel of the request.
4. View the SOAP response on the right panel.

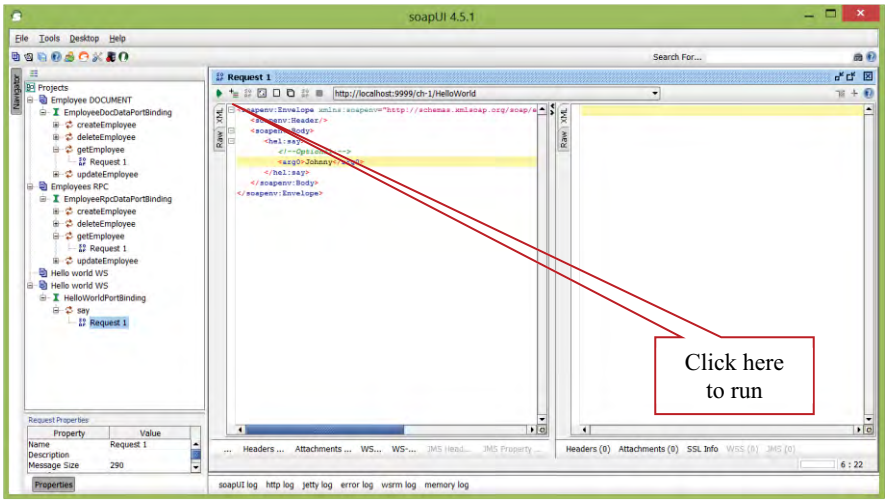


Figure 1-13 *Opening the HelloWorld WSDL*

To troubleshoot at the HTTP layer, click on the 'http log' button on the bottom of screen.

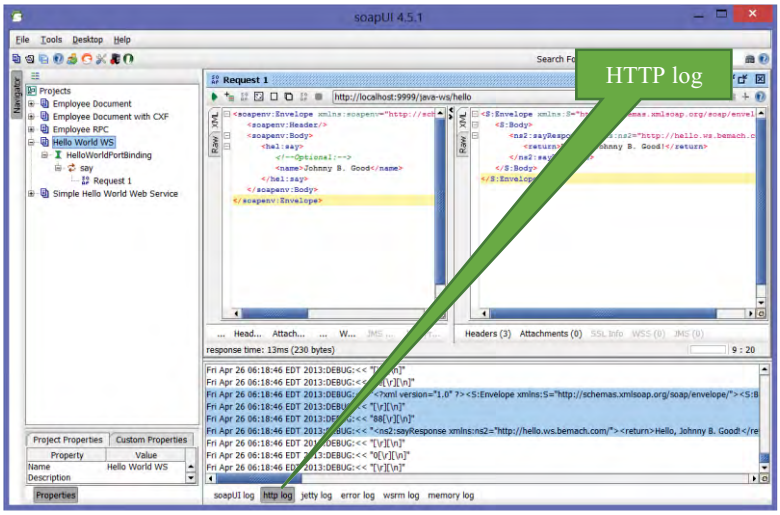


Figure 1-14 *Call an operation (method) of a Web Service*

1.7.3.1 SOAP Request:

The SOAP processor generates this request and sends it across the network to a WS invoking an operation *say* with a simple String argument.

Listing 1-6. A SOAP Request Message

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:hel="http://hello.ws.bemach.com/">
   <soapenv:Header />
   <soapenv:Body>
      <hel:say>
         <name>Johnny B. Good</name>
      </hel:say>
   </soapenv:Body>
</soapenv:Envelope>
```

1.7.3.2 SOAP response:

A SOAP response shows a simple return of a message string.

Listing 17. A SOAP Response Message

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:sayResponse xmlns:ns2="http://hello.ws.bemach.com/">
         <return>Hello, Johnny B. Good!</return>
      </ns2:sayResponse>
   </S:Body>
</S:Envelope>
```

1.8 Create a Web Service Client

A WS client code is simple to write; however, the amount of code required behind the scenes in order to ease the amount of coding on the client side can be substantial. Generated code enables a client application to call WS operations as it would normally do with another Java bean. This makes the programming of a client WS application a bit simpler. In the next chapter, we will use SAAJ APIs to create a Java client code that calls the HelloWorld Web Service.

The process of creating a Java Web Service client to call the HelloWorld Web Service involves the following steps:

1. Create a java-ws-client project in Eclipse.
2. Generate WS client stub from a service endpoint (http://localhost:9999/java-ws/hello?WSDL).
3. Write a Java client class.

1.8.1 Create a Project

1.8.2 Generate Web Service Stub

First, a generated WS client code is generated using a readily available tool, `wsimport`, from the Java JDK package. Second, a client code is written using the generated code.

1. Open a command prompt or a Unix terminal.
2. Go to the java-ws-client project directory.
3. Create a 'generated' directory.
4. Create a 'lib' directory.
5. Go to the 'generated' directory.
6. Run the following command:

```
wsimport -d . http://localhost:9999/java-ws/hello?wsdl
```

7. Package the generated client code:

```
jar cvf ../java-ws-generated.jar *
```

8. Move the java-ws-generated.jar file to the 'lib' directory.

1.8.3 Create Web Service Client

9. Return to Eclipse and refresh the Java project:
   a) Choose java-ws project.
   b) Choose File → Refresh.
   c) From project properties, choose Java Build Path/Libraries.
   d) Click on Add JARs and add the java-ws-generated.jar file.
   e) Click OK.

10. Create a new Java package: com.bemach.ws.hello.client.
11. Create a new Java class: HelloWorldClient.java.

Listing 1-8. A HelloWorld Web Service Client

```
package com.bemach.ws.hello.client;
/**
 * 2013 (C) BEM, Inc., Fairfax, Virginia
 *
 * Unless required by applicable law or agreed to in writing,
 * software distributed is distributed on an
 * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
 * KIND, either express or implied.
 *
 */

import java.net.MalformedURLException;
import java.net.URL;
import java.util.logging.Level;
import java.util.logging.Logger;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;

import com.bemach.ws.hello.HelloWorld;
import com.bemach.ws.hello.HelloWorldService;

/**
 * The following code is a normal way of going about to call
 * a web services using Java code.
 * It is much easier to comprehend.
 *
 */
public class HelloWorldWSClient {
   private static final Logger LOG = Logger.getLogger(HelloWorldWSClient.
class.getName());

   public static void main(String[] args) {
      HelloWorldWSClient client = new HelloWorldWSClient();
      try {
         client.say("Johnny B. Good");
      } catch (Exception e) {
         LOG.log(Level.SEVERE,"ERROR:"+e);
      }
   }

   public void say (String name) throws MalformedURLException {
      LOG.info("service ... ");

      QName qName = new QName("http://hello.ws.bemach.com/", "HelloWorldService");
      URL url = new URL("http://localhost:9999/java-ws/hello");
      Service service = HelloWorldService.create(url, qName);
      HelloWorld port = (HelloWorld)service.getPort(HelloWorld.class);

      String returnMsg = port.say(name);
      LOG.info("return: "+returnMsg);
   }
}
```

1.9 Run a Web Service Client

Output

```
Web Service response: Hello, Johnny B. Good!
```

1.10 References

Gottschalk, K., Graham, S., Kreger, H., & Snell, J. (2002). Introduction to Web Services architecture. *IBM Systems Journal, 41*(2), 170–177.

Kleijnen, S., & Raju, S. (2003). An Open Web Services Architecture. *Queue, 1*(1), 38–46.

Martin, J., Arsanjani, A., Tarr, P., & Hailpern, B. (2003). Web Services: Promises and Compromises. *Queue, 1*(1), 48–58.

2 SOAP

Objectives

After reading this chapter, you should:

1. Possess a basic understanding of SOAP
2. Be able to describe the structure of a SOAP message
3. Understand how to process a SOAP message
4. Be able to map a SOAP to a transport protocol
5. Be able to write a simple SOAP client using SOAP with Attachment for Java (SAAJ)

SOAP was an improvement of XML-RPC (Remote Procedure Call) that made it possible to use XML to represent data between two systems. Initially, SOAP message structure was relatively simple. Since SOAP used HTTP protocol, it was tunneled through firewall using the Internet infrastructure. According to World Wide Web Consortium (W3C), SOAP is defined as follows (http://www.w3.org/ TR/2007/REC-soap12-part1-20070427/):

> *…a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment. It uses XML technologies to define an extensible messaging framework providing a message construct that can be exchanged over a variety of underlying protocols. The framework has been designed to be independent of any particular programming model and other implementation specific semantics.*

SOAP was implemented as a plug-in to many middleware platforms and enabled data exchange over the Internet in addition to intranet. The initial emphasis on RPC thus allowed SOAP to become widely implemented as a wire-protocol over the Internet. Eventually, the need for interactions other than RPC led to the Documentation type of exchange. Protocols other than HTTP have also emerged over the years (e.g., SMTP and JMS).

SOAP Version 1.2 is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment (http://www.w3.org/TR/soap12-part1/#intro). SOAP specifications provide a formal set of conventions that governs how SOAP messages are generated and accepted by SOAP nodes (i.e., senders, receivers and intermediaries).

SOAP XML Payload

A SOAP message is enclosed in a SOAP envelope that contains a SOAP header and a SOAP body. The SOAP body is mandatory, whereas the SOAP header is optional. The SOAP message is the basic unit of communication between SOAP nodes. A SOAP node can transmit, receive or relay a SOAP message. The responsibility of the ultimate destination is to process SOAP messages according to the standard. A SOAP message traverses between a SOAP sender and a SOAP receiver. The message path may have SOAP intermediary nodes in a distributed computing environment. These SOAP intermediaries are thought of as 'men-in-the-middle', which can cause significant security problems.

The two major message exchange patterns that SOAP supports are 1) request-response (in-out) and 2) request (in-only). The first pattern is often used where a SOAP request is processed and a SOAP response is returned to the SOAP sender node. The second pattern is used when the SOAP sender has no interest in receiving a response (e.g., notification). A SOAP response may be returned in an asynchronous mode. For example, a server may take a long time to process a SOAP request. Instead of waiting for this request to be completed, a SOAP client may receive a callback when the server completes the processing.

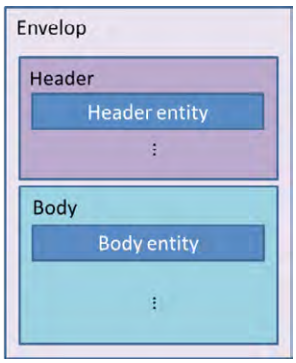


Figure 2-1. *SOAP Message Structure*

2.1 Examples of SOAP messages

The following shows SOAP messages in three forms: requests, responses and faults.

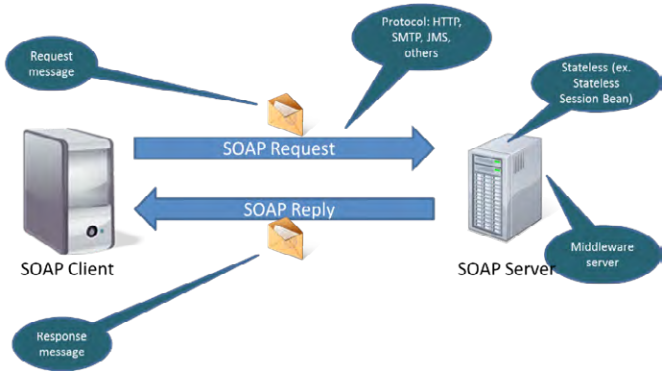


Figure 2-2 *SOAP message exchange*

Messages can be exchanged using HTTP protocol; thus, firewalls would allow the messages to pass through. SOAP messages can also ride naturally over a secure channel using HTTPS protocol as well. SOAP promotes a loose-coupling computing paradigm where the knowledge of software components is limited to interface of the service.

2.1.1 SOAP request message

The request represents an invocation of getEmployee method with an emplNo parameter of 100011. Note that the body contains a single body block with one XML element, <emp:getEmployee>. A request is marshalled (serialized) into an XML document prior to transport across the network to a remote SOAP server. When the message arrives at the server, it is 'un-marshalled' by a SOAP engine and the targeted method is invoked.

- A SOAP header is not used in this case.
- The SOAP body consists of a single element, emp:getEmployee.
- getEmployee: To get an employee record, an employee number (emplNo) is required.

Listing 2-1. A SOAP Message Request

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:emp="http://employees.ws.bemach.com/">
   <soapenv:Header />
   <soapenv:Body>
      <emp:getEmployee>
         <emplNo>100011</emplNo>
      </emp:getEmployee>
   </soapenv:Body>
</soapenv:Envelope>
```

2.1.2 SOAP response message

When the SOAP server successfully processes a request, a response is then marshaled into an XML document and returned as a part of HTTP response to the client. The client SOAP engine then unmarshals the message to further process the result of the call.

- A SOAP header is not used.
- The SOAP body has a single element, ns2:getEmployeeResponse. Note that namespace, ns2, is used and referenced.
- Within the SOAP body, an employee record is returned.

Listing 2-2. A SOAP Message Response

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:getEmployeeResponse xmlns:ns2="http://employees.ws.bemach.com/"
         xmlns:ns3="http://bemach.com">
         <return>
            <emplNo>100011</emplNo>
            <firstName>Shmuel</firstName>
            <lastName>Birge</lastName>
            <birthDate>1956-07-20T00:00:00-04:00</birthDate>
            <gender>M</gender>
            <hireDate>1989-11-23T00:00:00-05:00</hireDate>
         </return>
      </ns2:getEmployeeResponse>
   </S:Body>
</S:Envelope>
```

2.1.3 SOAP fault message

If a request contains an invalid employee number, then a SOAP fault message can be optionally returned. A SOAP fault may contain messages that help the client to resolve an error or unexpected condition. It must include a fault code, fault string, and a detailed error message.

Listing 2-3. A SOAP Fault

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <S:Fault xmlns:ns4="http://www.w3.org/2003/05/soap-envelope">
         <faultcode>S:Server</faultcode>
         <faultstring>No such employee!</faultstring>
         <detail>
            <ns2:SOAPException xmlns:ns2="http://employees.ws.bemach.com/"
               xmlns:ns3="http://bemach.com">
               <message>No such employee!</message>
            </ns2:SOAPException>
         </detail>
      </S:Fault>
   </S:Body>
</S:Envelope>
```

The basic assumption is that all data types are exchanged between SOAP nodes in XML format. In reality, not all data should be translated into XML (e.g., binary image, proprietary data formats). Thus, a proposed solution is to use SOAP attachments similar to the email protocol. There are problems with this approach, though, in that scalability can become an issue. Furthermore, not all SOAP engine implements SOAP attachment; thus, compatibility becomes an issue between SOAP nodes. This does not discount the overhead of an attachment.

2.2 Mapping SOAP to HTTP

The SOAP message exchange maps nicely into HTTP protocol. In this example, an HTTP POST is used for invocating a SOAP method on a server:

Listing 2-4. An HTTP Post

```
POST /java-ws/hello HTTP/1.1
Accept-Encoding: gzip,deflate
Content-Type: text/xml;charset=UTF-8
SOAPAction: ""
Content-Length: 298
Host: localhost:9999
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.1.1 (java 1.5)

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:hel="http://hello.ws.bemach.com/">
   <soapenv:Header/>
   <soapenv:Body>
      <hel:say>
         <!-Optional:->
         <name>Johnny B. Good</name>
      </hel:say>
   </soapenv:Body>
</soapenv:Envelope>
```

Listing 2-5. Another HTTP Post

```
POST /rpc/employees HTTP/1.1
Accept-Encoding: gzip,deflate
Content-Type: text/xml;charset=UTF-8
SOAPAction: ""
Content-Length: 290
Host: localhost:9999
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.1.1 (java 1.5)

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:emp="http://employees.rpc.ws.bemach.com/">
   <soapenv:Header/>
   <soapenv:Body>
      <emp:getEmployee>
         <emplNo>10002</emplNo>
      </emp:getEmployee>
   </soapenv:Body>
</soapenv:Envelope>
```

In this case, a return of an HTTP POST for a SOAP response is relatively simple:

Listing 2-6. An HTTP Response

```
HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: text/xml;charset="utf-8"

5e
<?xml version="1.0" ?><S:Envelope
xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"><S:Body>

88
<ns2:sayResponse xmlns:ns2="http://hello.ws.bemach.com/"><return>Hello,
Johnny B. Good!</return></ns2:sayResponse></S:Body></S:Envelope>

0
```

Listing 2-7. Another HTTP Response

```
HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: text/xml; charset=utf-8
Date: Thu, 11 Apr 2013 10:20:49 GMT

5e
<?xml version="1.0" ?><S:Envelope
xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"><S:Body>

167
<ns3:getEmployeeResponse xmlns:ns2="http://bemach.com"
xmlns:ns3="http://employees.rpc.ws.bemach.com/"><return><emplNo>10002</
emplNo><firstName>Bezalel</firstName><lastName>Simmel</
lastName><birthDate>1964-06-02T00:00:00-04:00</birthDate><gender>F</
gender><hireDate>1985-11-21T00:00:00-05:00</hireDate></return></
ns3:getEmployeeResponse></S:Body></S:Envelope>

0
```

2.3 SAAJ Client

A SOAP with Attachment API for Java (SAAJ) is more complex to write. It allows the developers to gain direct access to methods of creating SOAP messages. You can manipulate XML directly using Java APIs. Unlike the example in Chapter 1 of writing the client code that uses JAX-RPC, writing an SAAJ client can be laborious. SAAJ allows a small footprint and support for binary data; however, it is difficult to work with in various binary data formats.

The steps for creating a SOAP client using SAAJ are as follows:

1. Create a SOAP message that includes a SOAP header and SOAP body.
2. Add necessary elements to the SOAP header and SOAP body.
3. Create a SOAP connection (URL).
4. Send the SOAP message to the SOAP server.
5. Process the response message.

A SOAP client can be written in Java with a basic understanding of SOAP messages. A client code of the HelloWorld service is as follows.

Listing 2-8. HelloWorldSOAPClient.java class Using SAAJ

```
package com.bemach.ws.hello.client;

/**
 * 2013 (C) BEM, Inc., Fairfax, Virginia
 *
 * Unless required by applicable law or agreed to in writing,
 * software distributed is distributed on an
 * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
 * KIND, either express or implied.
 *
 */

import java.net.MalformedURLException;
import java.net.URL;
import java.util.Iterator;
import java.util.logging.Logger;

import javax.xml.soap.MessageFactory;
import javax.xml.soap.Name;
import javax.xml.soap.SOAPBody;
import javax.xml.soap.SOAPBodyElement;
import javax.xml.soap.SOAPConnection;
import javax.xml.soap.SOAPConnectionFactory;
import javax.xml.soap.SOAPElement;
import javax.xml.soap.SOAPException;
import javax.xml.soap.SOAPFactory;
import javax.xml.soap.SOAPHeader;
import javax.xml.soap.SOAPMessage;

/**
 * The following code is a complex way of go about calling a
 * Web services using Java code.
 *
 * There are times this is necessary.
 *
 */
public final class HelloWorldSOAPClient {
   private static final Logger LOG = Logger.getLogger(HelloWorldSOAPClient.
class.getName());
   /**
    *

      <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:m="http://hello.ws.bemach.com/">
         <soapenv:Header/>
         <soapenv:Body>
            <m:say>
               <arg0>Johnny</arg0>
            </m:say>
         </soapenv:Body>
      </soapenv:Envelope>
```

```
      <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
         <S:Body>
            <ns2:sayResponse xmlns:ns2="http://hello.ws.bemach.com/">
               <return>Hello, Johnny!</return>
            </ns2:sayResponse>
         </S:Body>
      </S:Envelope>

    */

   private HelloWorldSOAPClient() {}

   public static void main(String[] args) {
      try {
         // 1. Create a SOAP Message
         SOAPFactory sf = SOAPFactory.newInstance();
         MessageFactory mf = MessageFactory.newInstance();
         SOAPMessage sm = mf.createMessage();
         SOAPHeader sh = sm.getSOAPHeader();
         SOAPBody sb = sm.getSOAPBody();
         sh.detachNode();

         // 2. Add necessary elements to header and body.
         Name bodyName = sf.createName("say","m","http://hello.ws.bemach.com/”);
         SOAPBodyElement be = sb.addBodyElement(bodyName);
         Name name = sf.createName("name");
         SOAPElement arg0 = be.addChildElement(name);
         arg0.addTextNode("Johnny");

         // 3. Create a SOAP connection
         URL ep = new URL(String.format("http://localhost:%s/java-ws/
hello?WSDL",args[0]));
         SOAPConnectionFactory scf = SOAPConnectionFactory.newInstance();
         SOAPConnection sc = scf.createConnection();

         // 4. Send a SOAP message using the connection.
         SOAPMessage response = sc.call(sm, ep);
         sc.close();

         // 5. Process the response message
         SOAPBody respBody = response.getSOAPBody();
         Iterator it = respBody.getChildElements();
         SOAPBodyElement el = (SOAPBodyElement)it.next();
         LOG.info("resp="+el.getNodeName());

         it = el.getChildElements();
         SOAPElement ret = (SOAPElement) it.next();
         LOG.info(String.format("%s=%s",ret.getNodeName(),ret.getTextContent()));
      } catch (UnsupportedOperationException e) {
         LOG.severe(e.getMessage());
      } catch (SOAPException e) {
         LOG.severe(e.getMessage());
      } catch (MalformedURLException e) {
         LOG.severe(e.getMessage());
      }
   }
}
```

SAAJ has many more capabilities that you can use to work with SOAP messages. For more information on this topic, see the references at the end of this chapter.

SOAP messages can be structured in two ways: document and RPC styles. At first, only RPC style was supported. For the RPC model, the SOAP body defines a specific method with associated parameters that the client can invoke. Thus, the message exchange between client and server can be restricted. Method calls are tied directly with the in and out parameters.

Unlike the RPC style, however, the document style enables the client and server to exchange messages in whatever formats they choose. The SOAP body contains messages that do not follow any SOAP formatting rules. The body can be validated against a schema.

2.4 Summary

SOAP is the foundation of Web Services. SOAP remains a dominant message exchange protocol used for B2B integration. Note that no security concern is deliberately mentioned in SOAP. SOAP is mainly about exchanging messages between two systems. It promotes loose-coupling computation. Security and other capabilities are left out to avoid complexity in the protocol.

In the next chapter, we will discuss how Web Services use SOAP as a transport protocol to promote the service-oriented paradigm.

2.5 References

Additional information can be found in the following documents, which are available online:

Albrecht, C.C. (2004). "How clean is the future of SOAP?" Commun. ACM **47**(2): 66–68.

SOAP Version 1.2 Part 0: Primer (Second Edition), Retrieved August 27, 2007 from: http:// www.w3.org/TR/2007/REC-soap12-part0-20070427/

Bong, G. (2002). Apache SOAP type mapping, Part 1: Exploring Apache's serialization APIs, Retrieved August 27, 2007 from: http://www-106.ibm.com/developerworks/webservices/ library/ws-soapmap1/

Bong, G. (2002). Apache SOAP type mapping, Part 2: A serialization cookbook, Retrieved August 27, 2007 from: http://www-106.ibm.com/developerworks/webservices/library/ws-soapmap2/

Cohen, F. (2001). Myths and misunderstandings surrounding SOAP, Retrieved August 27, 2007 from: http://www-106.ibm.com/developerworks/library/ws-spmyths.html

Marchal, B. (2001). Soapbox: Why I'm using SOAP, Retrieved August 27, 2007 from:
http://www-106.ibm.com/developerworks/xml/library/x-soapbx1.html

McLaughlin, B. (2001). Soapbox: Magic bullet or dud?, Retrieved August 27, 2007 from:
http://www-106.ibm.com/developerworks/library/x-soapbx2/index.html

McLaughlin, B. (2001). Soapbox: Industrial strength or suds?, Retrieved August 27, 2007 from:
http://www-106.ibm.com/developerworks/library/x-soapbx3/index.html

W3C SOAP Tutorial, Retrieved November 5, 2010 from:
http://www.w3schools.com/soap/default.asp

3 Web Service Description Language (WSDL)

Objectives

After reading this chapter, you should:

1. Possess a basic understanding of a WSDL
2. Be able to decipher a WSDL

WSDL assists service clients that need to know how to bind a service automatically. A service contract must be established between the service consumer and provider. A published WSDL describes in detail the contract, which may include messages, operations, bindings and locations of the service.

When a Web Service is ready for use, its location and access are made known to external systems. WSDL is based on the Interface Description Language (IDL), which describes the interface of a software component for other components to use. In RPC, a developer defines an interface of a component to be exposed to external applications that do not share the same language.

Once an interface is described, in most cases, a tool is used to generate client and server stubs for the client side and the server side, respectively, to use. The server application uses the server stub for its implementation of the service, while the client application uses the client stub for its service invocation.

3.1 WSDL structure

WSDL consists of two parts: abstract interface and concrete implementation. While the abstract interface describes the operations and messages of a service, the concrete implementation part binds the abstract interface with a concrete network address. Thus, the two together comprise a service.

A WSDL is an XML document with a root element named *definitions*. The definition includes two parts: abstract and concrete. The abstract descriptions consist of *types, message,* and *portType.* The concrete part consists of *binding*, and *service.* Abstract parts describe the operations that are independent of way in which the service will be implemented, while the concrete part specifies the protocol and location of the service where it can be invoked.

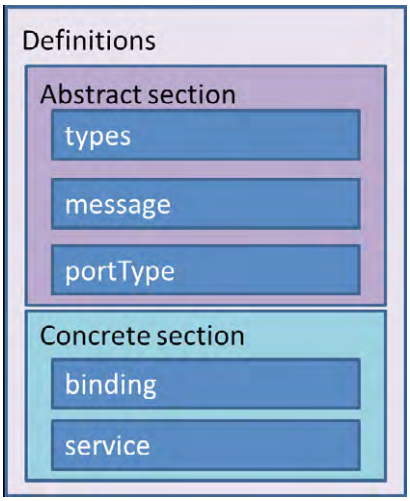


Figure 3-1. *WSDL structure*

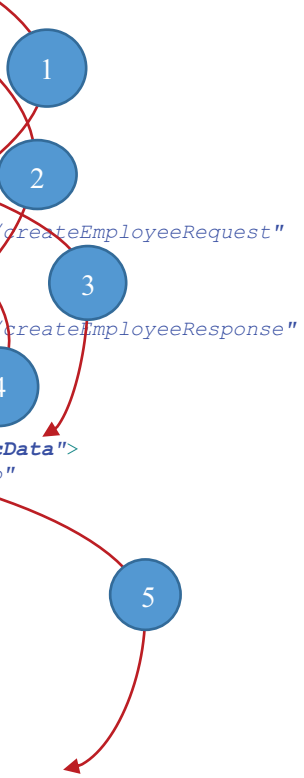
In the following section, we will examine the structure of a WSDL more closely to understand the relationships between the abstract interface and the concrete implementation.

Listing 3-1 Sample WSDL

```
<definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://hello.ws.bemach.com/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
targetNamespace="http://hello.ws.bemach.com/"
   name="HelloWorldService">
   <types>
      <xsd:schema>
         <xsd:import namespace="http://hello.ws.bemach.com/"
            schemaLocation="http://localhost:9999/java-ws/hello?xsd=1" />
      </xsd:schema>
   </types>
   <message name="say">
      <part name="parameters" element="tns:say" />
   </message>
   <message name="sayResponse">
      <part name="parameters" element="tns:sayResponse" />
   </message>
   <portType name="HelloWorld">
      <operation name="say">
         <input message="tns:say" />
         <output message="tns:sayResponse" />
      </operation>
   </portType>
   <binding name="HelloWorldPortBinding" type="tns:HelloWorld">
      <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
         style="document" />
      <operation name="say">
         <soap:operation soapAction="" />
         <input>
            <soap:body use="literal" />
         </input>
         <output>
            <soap:body use="literal" />
         </output>
      </operation>
   </binding>
   <service name="HelloWorldService">
      <port name="HelloWorldPort" binding="tns:HelloWorldPortBinding">
         <soap:address location="http://localhost:9999/java-ws/hello" />
      </port>
   </service>
</definitions>
```

Listing 3-2. Another Sample WSDL

```
<definitions
   xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-
1.0.xsd"
   xmlns:wsp="http://www.w3.org/ns/ws-policy"
xmlns:wsp1_2="http://schemas.xmlsoap.org/ws/2004/09/policy"
   xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://employees.doc.ws.bemach.com/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
targetNamespace="http://employees.doc.ws.bemach.com/"
   name="EmployeeDocDataService">
   <types>
      <xsd:schema>
         <xsd:import namespace="http://employees.doc.ws.bemach.com/"
            schemaLocation="http://localhost:9999/doc/employees?xsd=1" />
      </xsd:schema>
      <xsd:schema>
         <xsd:import namespace="http://bemach.com"
            schemaLocation="http://localhost:9999/doc/employees?xsd=2" />
      </xsd:schema>
   </types>
   <message name="createEmployee">
      <part name="parameters" element="tns:createEmployee" />
   </message>
   <message name="createEmployeeResponse">
      <part name="parameters" element="tns:createEmployeeResponse" />
   </message>

   <message name="SOAPException">
      <part name="fault" element="tns:SOAPException" />
   </message>
   <portType name="EmployeeDocData">
      <operation name="createEmployee">
         <input

   wsam:Action="http://employees.doc.ws.bemach.com/EmployeeDocData/createEmployeeRequest"
            message="tns:createEmployee" />
         <output

   wsam:Action="http://employees.doc.ws.bemach.com/EmployeeDocData/createEmployeeResponse"
            message="tns:createEmployeeResponse" />
      </operation>

   </portType>
   <binding name="EmployeeDocDataPortBinding" type="tns:EmployeeDocData">
      <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
         style="document" />
      <operation name="createEmployee">
         <soap:operation soapAction="" />
         <input>
            <soap:body use="literal" />
         </input>
         <output>
            <soap:body use="literal" />
         </output>
      </operation>
   </binding>
   <service name="EmployeeDocDataService">
      <port name="EmployeeDocDataPort" binding="tns:EmployeeDocDataPortBinding">
         <soap:address location="http://localhost:9999/doc/employees" />
      </port>
   </service>
</definitions>
```



3.2 WSDL Interface

The service interface is defined using the portType element. This element contains a list of operation elements that make up a Web Service. Each operation consists of one in and one out message, and optionally one fault message. All operations exchange messages between a client and a server. A message is an abstract data type that contains the data.

Refer to listings 3-1 and 3-2 above, which highlight the elements of a basic WSDL document. The five major components of the WSDL and their relationships are visible.

3.2.1 <types> element

All data types are defined inside this element types. Usually, the element types points to an external URL that contains an XML schema which may contain other schemas. WSDL specification does not prohibit the use of other type definition systems; however, it prefers the XML schema.

3.2.2 <message> element

A message element is an abstract data type describing the input and output of an operation. It consists of parts that are linked to types. Each operation (@WebMethod) may contain three message types: input, output, and fault. Each of these message types is unique throughout the entire WSDL document. Each has a unique name. Inside each message, the parts define the actual types that are properly defined inside the <types> element specifically (RPC style) or via a schema (Document style).

3.2.3 <portType> element

The <portType> element consists of an abstract set of operations. These operations may be supported by one or more endpoints. Operations are used to exchange messages between a service consumer and the provider. Message exchange protocol can take one of the following patterns:

- One-way – the service endpoint receives a message
- Request-response – the service endpoint receives a message and sends a correlated message
- Solicit-response – the service endpoint sends a message and receives a correlated message
- Notification – the service endpoint sends a message

In the example above (Listing 3-2), the WS consists of one operation (i.e., createEmployee) that has two messages – one input and one output message. The input message is named 'tns:createEmployee' and the output message 'tns:createEmployeeResponse'. These two messages can be traced back to the message elements. Think of portType as being like a Java class where methods and attributes are defined.

3.2.3.1 <operation> element

A service may have one or more operations (or methods). Each operation is independently defined with one input message, one output message, and an optional fault message. These messages are defined using the message element. Similar to a Java class definition, each operation defines a behavior of a service.

3.3 WSDL Implementation

The <binding> element of the WSDL connects the WSDL abstract interface to concrete implementation. In this section, we will pay specific attention to the <binding> element where the connection takes place. Thus, in some way, <binding> is the central element of the entire WSDL specification. This is where the two worlds meet.

3.3.1 <binding> element

Binding maps <portType> to a implementation specified in the <service> element. From listing 3-1, <portType> HelloWorld is bound to <port> HelloWorldPort inside the <service> element. The implementation of HelloWorld is SOAP over HTTP (<soap:binding>). SOAP specifies data formats and HTTP is a specific protocol to be used for the service offering.

Binding does not specify any particular language in its implementation. The way in which the service is implemented is beyond the scope of the WSDL.

3.3.2 <service> element

A service is a collection of network-specific addresses (<port>) where the service may be rendered. <port> and <portType> are linked via a <binding> element. The connection is critical for runtime dynamic binding between the service consumers and providers.

The <port> element describes the network address that enables a service consumer to interact with the service being offered. A service with multiple ports is possible; thus, the choice is left to the consumer. In the sample WSDL, only one port is provided and its location is specified as http://localhost:9999/java-ws/hello. Note that the host name and port number can be modified at runtime to point to whichever server is hosting the service.

The following describes the linkages between the major elements of a WSDL.

Figure 3-2 *Linkages inside WSDL*

3.4 References

Flaherty, B. (2004). "WSDL: Defining Web Services." Intercom 51(8): 26–28.

Lessen, T., J. Nitzsche, et al. (2009). "Conversational Web Services: Leveraging BPEL (light) for expressing WSDL 2.0 message exchange patterns." Enterprise Information Systems 3(3): 347–367.

Web Services Description Language (WSDL) 1.1, Retrieved August 27, 2007 from: http://www.w3.org/TR/wsdl

WSDL Tutorial, Retrieved November 7, 2007 from: http://www.w3schools.com/wsdl/default.asp

4 A Sample Web Service Application

Objectives

After completing this chapter, you should be able to:

1. Write a Web Service that provides access to employee records stored in a relational database
2. Write a Web Service using JDK 6 or above
3. Publish a Web Service using basic Java Endpoint class
4. Test a Web Service with SOAPUI testing tool
5. Use wsimport to generate a Web Service stub for the client
6. Write a simple Web Service consumer to invoke a Web Service

4.1 A Sample application

Welcome to the world of Web Services! You may find this chapter technically challenging at first; however, as you work your way through the examples, you will find that the same patterns are used repeatedly throughout. If you think of writing Web Services as similar to writing any other Java class, that may help to ease any anxiety about the difficulty of this task.

In this application, we deploy a simple SOAP server using basic Java JDK delivery. In order to make this application work, you will need the following software packages that can be downloaded from the Internet (more instructions are included in Appendix A).

- Java JDK 6
- MySQL Community Server 5.6
- MySQL Employees sample database
- MySQL JDBC driver

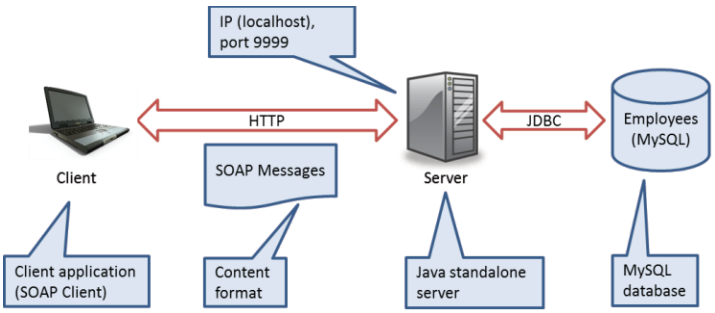


Figure 4-1. *A n-tier application*

A WS application is created using a Java framework to enable a WS consumer to manipulate employee records stored in a relational database. Accessing the database from a Web server is accomplished using JDBC technology. MySQL is the relational database used for this example. The transport protocol for the WS is HTTP.

To avoid adding complexity to an already complicated concept, security concerns are not considered in this example. Accessing the database from a remote machine (i.e., WS client) without proper authentication is not a good practice; however, in this application, accessing the database with fixed user ID and password is a matter of simplicity, not security. Furthermore, the use of the data source is much more efficient using direct JDBC calls, however, the sample code does not follow that standard convention.

The basic Java Endpoint class does not scale well in a business computing environment, but it is used here to allow the simplest Java environment capable of supporting a simple WS application. In later chapters, you can apply similar programming principles and techniques for WS programming to deploy WS applications on an Apache Tomcat or an Oracle WebLogic server. These two servers are covered in Chapters 5 and 6, respectively.

Remember, the central idea of this chapter in terms of WS programming is how to get data from the database through the use of WS technology, and SOAP in particular.

4.1.1 Use Case Diagram

Consider the following use case diagram for this sample application. From the perspectives of WS clients, it invokes four operations of an employee data service. Basic data exchange includes two major data types: employee number and employee record.

A resource can be created, read (or obtained), updated (or changed), and deleted. The concept of CRUD has been fundamental to computer programming since the beginning of the field computer science. We have a set of employee records stored in a database, and we want to manipulate them from a remote machine using SOAP via WS technology.

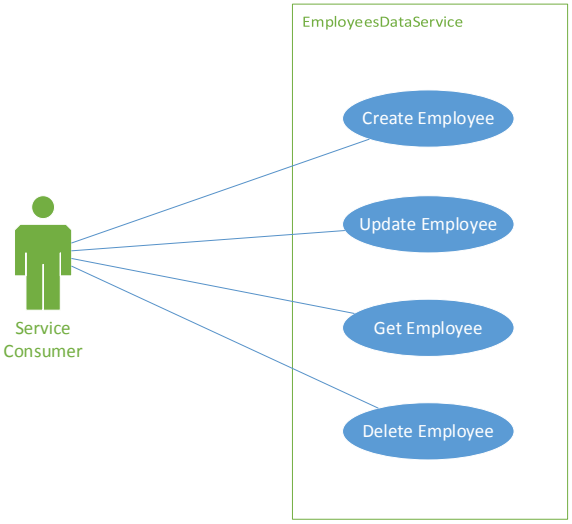


Figure 4-2. *Use Cases*

Use Case 1: Create Employee
Primary Actor: Service Consumer
Main Success Scenario:

1. An end-user enters required employee information.
2. The service consumer then verifies the information.
3. The service consumer then calls the *employees* data service to create a new employee.
4. The service consumer presents a new employee number to the end-user.

Use Case 2: Update Employee
Primary Actor: Service Consumer
Main Success Scenario:

1. An end-user updates the required employee information.
2. The service consumer then verifies the information.
3. The service consumer then calls the *employees* data service for the update.
4. The service consumer informs the end-user about the status of the update.

Use Case 3: Get (Read) Employee
Primary Actor: Service Consumer
Main Success Scenario:

1. An end-user enters an employee number.
2. The service consumer then validates the number.
3. The service consumer then calls the *employees* data service to retrieve the employee record.
4. The service consumer presents the employee record to the end-user.

Use Case 4: Delete Employee
Primary Actor: Service Consumer
Main Success Scenario:

1. An end-user enters an employee number.
2. The service consumer then validates the number.
3. The service consumer then calls the *employees* data service to remove the employee record.
4. The service consumer informs the end-user about the status of the deletion.

A SOAP exception is thrown in when an error condition occurs.

4.1.2 Sequence Diagram

In a typical WS call, many layers of software are involved; however, at a high level, the sequence of actions may be represented as in the following sequence diagram.

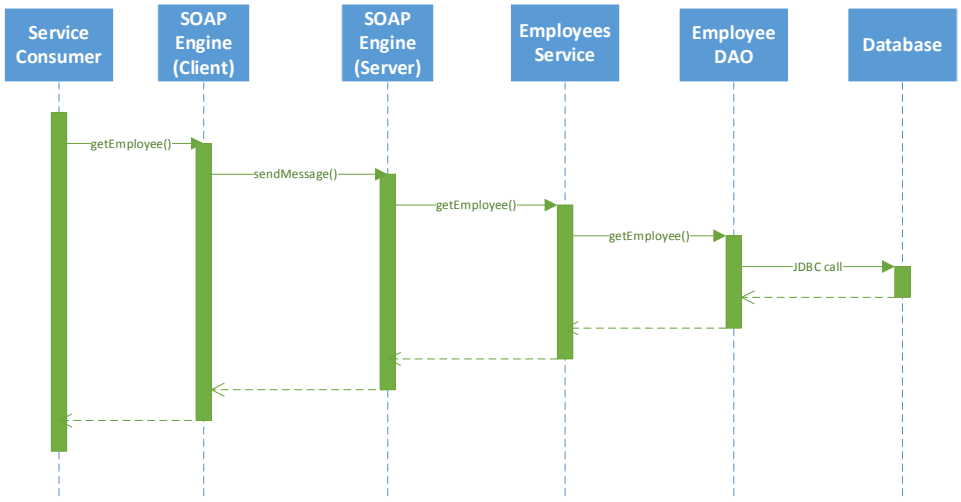


Figure 4-3. *Sequence diagram of a getEmployee operation*

Since all four operations are the basic request-response type of message exchange, a single sequence diagram of getEmployee operation is shown.

A consumer, when ready, invokes a SOAP engine on the client side to begin a SOAP call across the network. In this scenario, the consumer understands the SOAP message fully and constructs a SOAP message using SOAP with Attachment API for Java (SAAJ). Once a SOAP message is formed, the SOAP engine sends the message to the remote server via HTTP. After successfully receiving the message, the server processes the request by invoking the appropriate business or data services in the backend. In this case, the getEmployee method of the *employees* service is invoked. Before the data access layer is called, additional business logic processing can be done in this class to manipulate the data. EmployeeDAO is a component that interacts directly with the database using JDBC for data processing. The data source may not always be a relational database.

Once the processing is completed, the employees service, with the help of the WS package, forms a SOAP message and returns to the SOAP engine on the server side. As a part of the request-response message exchange pattern, the response is then returned to the SOAP engine on the client side. Once the client SOAP engine successfully receives the message, it returns to the Service Consumer for final processing.

The process of forming a SOAP message is often called 'marshalling'. Conversely, the process of decoding a SOAP message into a native form for further processing is called 'unmarshalling'.

This sequence diagram shows an example of a synchronous message exchange. In other words, activities in this diagram occur in sequence. In some cases, the processing may take a long time, and the server may return immediately before the processing completes. This is a form of asynchronous message exchange. When the server has completed processing the request, it may initiate a call to the client to return the response with the actual data or simply a notification. The client can also periodically poll the server for data. The second option suffers two problems. If the timing window between two polls is too large, the delay can be significant. If it is small, it wastes valuable processing power on both sides.

4.1.3 Deployment Diagram

The simple deployment of this WS application is depicted as follows:

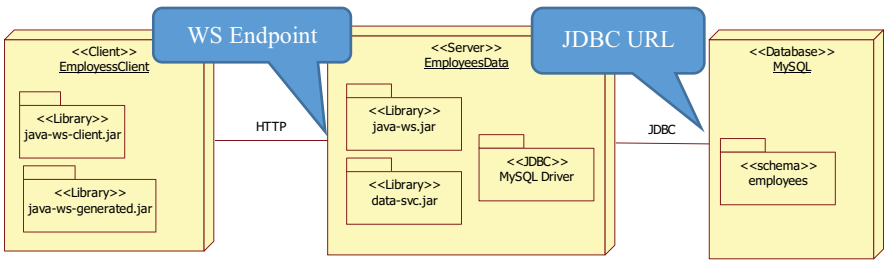


Figure 4-4. *A Simple Deployment Diagram*

In a real-world application, an IT organization may deploy a complex mesh of servers and databases to manage their WS activities. There can be many client applications. In this sample WS application, we create an environment that includes a client machine, a server machine, and a database machine. These machines can be virtual, which means that all three can be hosted by a single physical machine.

On the server side, we develop two sets of Java libraries – java-ws.jar and data-svc.jar. The first contains the WS code that interacts with the client over the network protocol HTTP. The second deals with the database access via JDBC calls with the help of MySQL driver code written by MySQL database developers. Together, they comprise a complete application.

On the client side, we develop a Java library that contains the WS client code, java-ws-client.jar. We use wsimport to generate the second library, java-ws-generated.jar. This second library contains all of the necessary code to interact with the server WS engine.

4.1.4 JDBC URL

To access a relational database from a Java application, a database connection must be established using a JDBC URL with the following format:

jdbc:<subprotocol>:<subname>

where

- <subprotocol> is the name of the driver that was registered with Oracle. In this application, 'mysql' is used.
- <subname> is the identification of the resource. It has the following format:

  //host:port/subsubname

  subsubname consists of the database schema name, user identification and password.

In this example, a full JDBC URL can be written as:

jdbc:mysql://localhost:3306/employees?user=empl_1&password=password

In order to access the database via JDBC connection, a database account was created and assigned to the *employees* database. It has all privileges to the *employees* database. The MySQL default access port is 3306. Before you run the SOAP server program, make sure to download a JDBC driver and include the driver in your Java's classpath.

4.1.4.1 DbConfig.java

In this sample code, the default values to connect to the MySQL database are (see DbConnection.java):

Hostname	Saintmonica
Port number	3306
Account	empl_1
Password	Password
Database name	Employees
JDBC driver name	com.mysql.jdbc.Driver
Subprotocol	Mysql

Table 1. *Database Configuration Parameters*

Listing 41. DbConfig.java class

```
package com.bemach.data;

/**
 * 2013 (C) BEM, Inc., Fairfax, Virginia
 *
 * Unless required by applicable law or agreed to in writing,
 * software distributed is distributed on an
 * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
 * KIND, either express or implied.
 *
 */

/**
 * Make sure to download MySQL JDBC Driver from the website
 * Extract it, and include this file (name may be changed between
 * release): mysql-connector-java-5.1.24-bin.jar into your classpath
 *
 */
public class DbConfig {
   private String subprot = "mysql";
   private String host = "saintmonica";
   private String port = "3306";
   private String db = "employees";
   private String uid = "empl_1";
   private String psw = "password";
   private String driverName = "com.mysql.jdbc.Driver";

   public String getSubprot() {
      return subprot;
   }

   public void setSubprot(String subprot) {
      this.subprot = subprot;
   }

   public String getDriverName() {
      return driverName;
   }

   public void setDriverName(String driverName) {
      this.driverName = driverName;
   }

   public String getHost() {
      return host;
   }

   public void setHost(String host) {
      this.host = host;
   }
```

```
    public String getPort() {
        return port;
    }

    public void setPort(String port) {
        this.port = port;
    }

    public String getDb() {
        return db;
    }

    public void setDb(String db) {
        this.db = db;
    }

    public String getUid() {
        return uid;
    }

    public void setUid(String uid) {
        this.uid = uid;
    }

    public String getPsw() {
        return psw;
    }

    public void setPsw(String psw) {
        this.psw = psw;
    }
}
```

For the application, the DbConfigure class is a placeholder for all necessary configuration parameters for connecting to to the MySQL database.

4.1.5 Web Service Endpoint

A WS must be published via a unique service endpoint in order to be accessed by a WS client. A URL is a pointer to an available resource. This unique service endpoint can be stated using a URL with the following format:

<scheme>:<hier-part>?query

where

- <scheme> is http protocol
- <hier-part> is //host:port/path

In this example, the service endpoint is defined as:

http://localhost:9999/doc/employees

and

http://localhost:9999/rpc/employees

4.1.5.1 SvrConfig.java

For HTTP connectivity to be used for SOAP, the sample code must use the following default values:

Hostname	locahost
Port number	9999
Protocol	http

Table 2. *Server Configuration Parameters*

Listing 4-2. SvrConfig.java class

```
package com.bemach.ws.server;

/**
 * 2013 (C) BEM, Inc., Fairfax, Virginia
 *
 * Unless required by applicable law or agreed to in writing,
 * software distributed is distributed on an
 * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
 * KIND, either express or implied.
 *
 */

import com.bemach.data.DbConfig;

/**
 * Make sure to download MySQL JDBC Driver from the website
 * Extract it, and include this file (name may be changed between
 * release): mysql-connector-java-5.1.24-bin.jar into your classpath
 *
 */
public class SvrConfig {
   private String listenHostname = "localhost";
   private String listenPort = "9999";
   private String listenInterface = "HelloWorld";
   private String listenProtocol = "http";
   private DbConfig dbCfg = new DbConfig();

   public DbConfig getDbCfg() {
      return dbCfg;
   }

   public void setDbCfg(DbConfig dbCfg) {
      this.dbCfg = dbCfg;
   }

   public String getListenHostname() {
      return listenHostname;
   }

   public void setListenHostname(String listenHostname) {
      this.listenHostname = listenHostname;
   }

   public String getListenPort() {
      return listenPort;
   }

   public void setListenPort(String listenPort) {
      this.listenPort = listenPort;
   }

   public String getListenInterface() {
      return listenInterface;
   }
```

```
    public void setListenInterface(String listenInterface) {
        this.listenInterface = listenInterface;
    }

    public String getListenProtocol() {
        return listenProtocol;
    }

    public void setListenProtocol(String listenProtocol) {
        this.listenProtocol = listenProtocol;
    }

}
```

The SvrConfig class consists of information that is used to form a service endpoint for both styles – document and RPC. Furthermore, the class contains the configuration parameters that the data access code uses in order to access the database.

4.1.6 About the employees[1] sample database from MySQL

The employees database is a sample database from MySQL. The database schema was developed by professor Chua Hock Chuan at Nanyan Technological University in Singapore. The site cat be visited at http://www.ntu.edu.sg/home/ehchua/programming/sql/SampleDatabases.html.

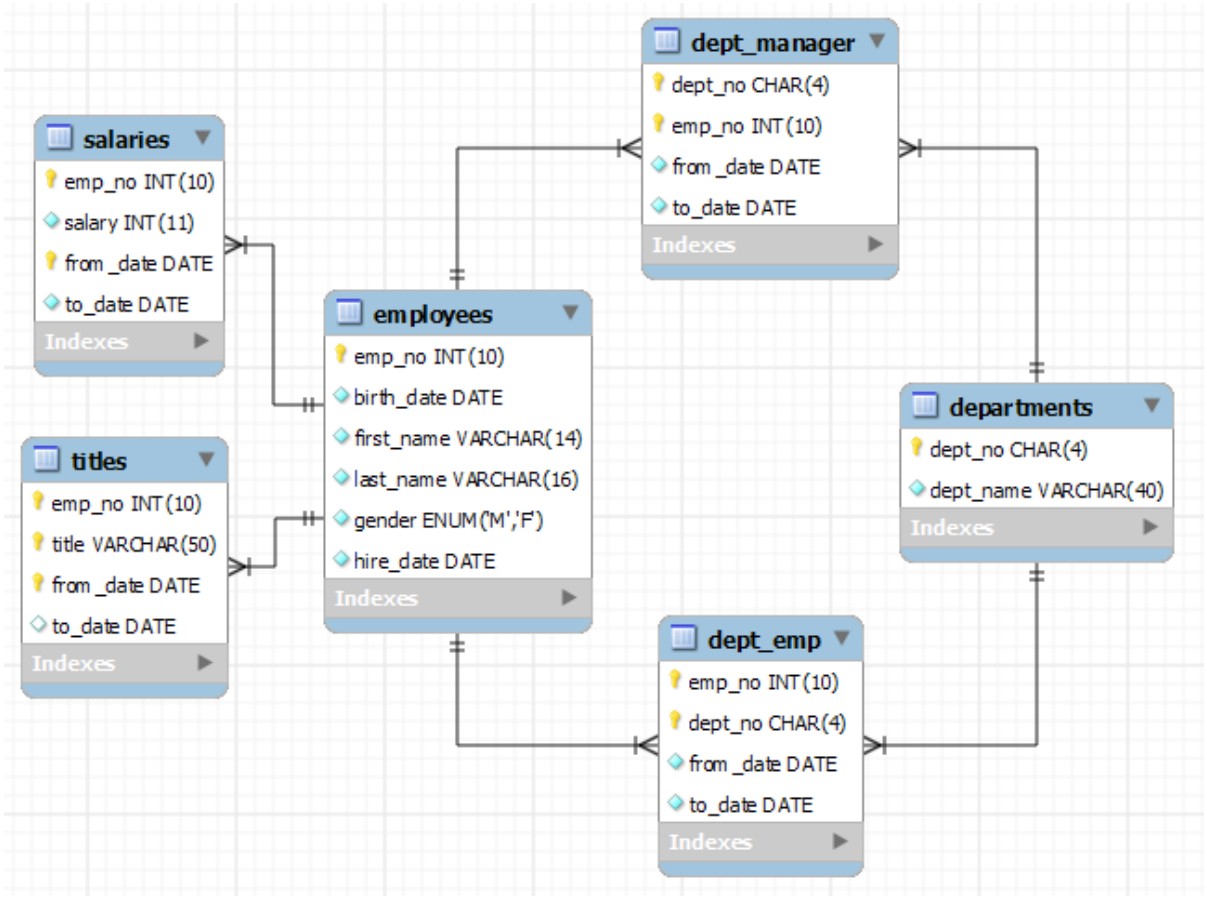


Figure 4-5. *Database Schema (Chua Hock Chuan)*

In this sample application, we use only the employees table. This table can be created using the following DDL:

Listing 4-3. Employees Table Definition

```
CREATE TABLE employees (
    emp_no      INT             NOT NULL,
    birth_date  DATE            NOT NULL,
    first_name  VARCHAR(14)     NOT NULL,
    last_name   VARCHAR(16)     NOT NULL,
    gender      ENUM ('M','F')  NOT NULL,
    hire_date   DATE            NOT NULL,
    PRIMARY KEY (emp_no)
);
```

All fields of the employees table are required with the primary key being the employee number. The employees data record can be represented by Employee class in Java. This class is defined as follows:

4.1.6.1 Employee.java

We create an Employee data object that contains an employee record. Employee is a Java class that uses Java Architecture for XML Binding (JAXB) annotations to assist the marshalling process. JAXB allows Java developer to use Java API to read and write objects to and from an XML document. It eases the process of reading and writing XML documents in Java. In particular, the annotation provides a simpler mechanism for the SOAP engine to transform Java objects into XML and vice versa.

Listing 4-4. Employee.java Class

```
package com.bemach.data;
/**
 * 2013 (C) BEM, Inc., Fairfax, Virginia
 *
 * Unless required by applicable law or agreed to in writing,
 * software distributed is distributed on an
 * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
 * KIND, either express or implied.
 *
 */

import java.io.Serializable;
import java.util.Calendar;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;

@XmlRootElement(name="EmployeeService",namespace="http://bemach.com")
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name="employee")

public class Employee implements Serializable{
   private static final long serialVersionUID = 1L;
   @XmlElement(required=true)
   private long emplNo;
   @XmlElement(required=true)
   private String firstName;
   @XmlElement(required=true)
   private String lastName;
   @XmlElement(required=true)
   private Calendar birthDate;
   @XmlElement(required=true)
   private String gender;
   @XmlElement(required=true)
   private Calendar hireDate;

   public long getEmplNo() {
      return emplNo;
   }
   public void setEmplNo(long emplNo) {
      this.emplNo = emplNo;
   }
   public String getFirstName() {
      return firstName;
   }
   public void setFirstName(String firstName) {
      this.firstName = firstName;
   }
```

```
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public Calendar getBirthDate() {
        return birthDate;
    }
    public void setBirthDate(Calendar birthDate) {
        this.birthDate = birthDate;
    }
    public String getGender() {
        return gender;
    }
    public void setGender(String gender) {
        this.gender = gender;
    }
    public Calendar getHireDate() {
        return hireDate;
    }
    public void setHireDate(Calendar hireDate) {
        this.hireDate = hireDate;
    }

}
```

All required fields are reflected in XML elements within the sequence. Optional elements often include `numOccurs="0"`. The Java data types are mapped neatly into XML intrinsic data types, as shown in the schema.

Listing 4-5. Data type of 'employee' within XSD

```
<xs:schema xmlns:tns="http://employees.rpc.ws.bemach.com/"
    xmlns:xs="http://www.w3.org/2001/XMLSchema" version="1.0"
    targetNamespace="http://employees.rpc.ws.bemach.com/">
    <xs:element name="SOAPException" type="tns:SOAPException" />
    <xs:complexType name="employee">
        <xs:sequence>
            <xs:element name="emplNo" type="xs:long" />
            <xs:element name="firstName" type="xs:string" />
            <xs:element name="lastName" type="xs:string" />
            <xs:element name="birthDate" type="xs:dateTime" />
            <xs:element name="gender" type="xs:string" />
            <xs:element name="hireDate" type="xs:dateTime" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="SOAPException">
        <xs:sequence>
            <xs:element name="message" type="xs:string" minOccurs="0" />
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

4.2 Develop a Web Service

A bottom-up approach for developing a Web Service involves the following activities:

- Write a data access object.
- Write a business logic object.
- Write a service object.
- Deploy a service to a server.
- Publish the server for use.

In this application, no business services are included, thus the activities are simplified as follows:

Listing 4-6. *Activities for writing Web Services with Java*

We create two Java projects under Eclipse – data-svc and java-ws. The data-svc project holds Java code that interacts with the database via JDBC. This project creates a library called 'data-svc.jar'. This library contains four Java classes:

- DbConfig.java
- DbConnection.java
- Employee.java
- EmployeeDao.java

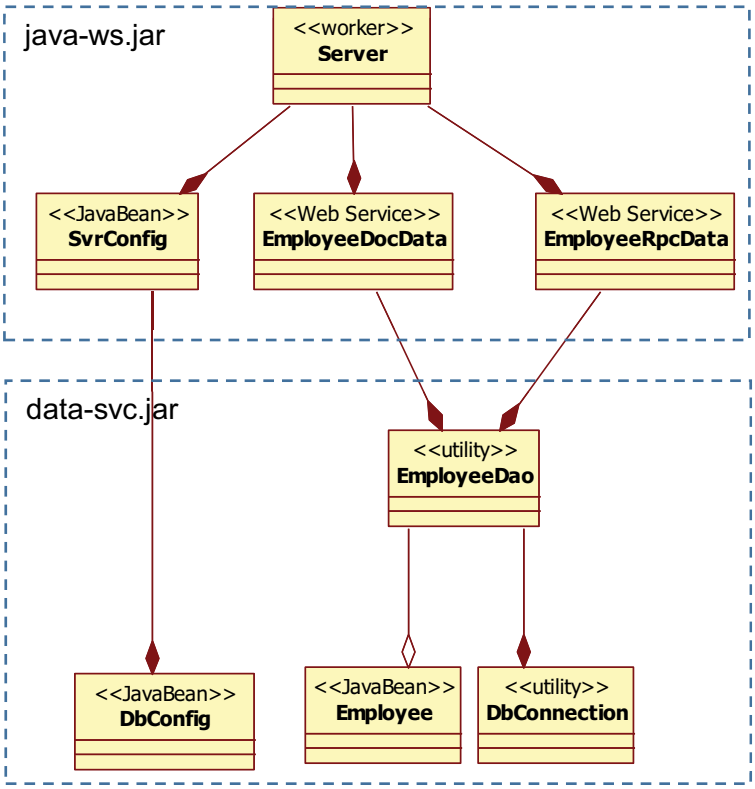
The java-ws project, which resulted in a java-ws.jar library, consists of the following Java classes:

- SvrConfig.java
- Server.java
- EmployeeDocData.java
- EmployeeRpcData.java

4.2.1 Class Diagram

A static view of the server application is depicted in the following class diagram.

Listing 4-7. A class diagram



In a class diagram, the hollow-diamond adornment indicates a part-whole relationship between the classes. This is called an 'aggregation'. On the other hand, the solid-diamond adornment represents a composite relationship between the classes. A composition is stronger than an aggregation in that the former involves a complete management of the lifetime of the object. For example, at runtime, an EmployeeDao object is responsible for allocation and deallocation of the DbConnection object. The Employee object is allocated by the EmployeeDao but deallocated by the EmployeeDocData or EmployeeRpcData object.

The dotted-line boxes indcate the boundaries of the two libraries to be created for this application.

4.2.2 Write Data Access Class

The Data Access Object (DAO) design pattern is used to provide abstract and encapsulated access of data from the data sources. It manages the connection with the data source to store and retrieve data.

First, we create a Java project called 'data-svc' (see section 7.2.1). After we complete our coding of the Java classes, this project should appear as follows:

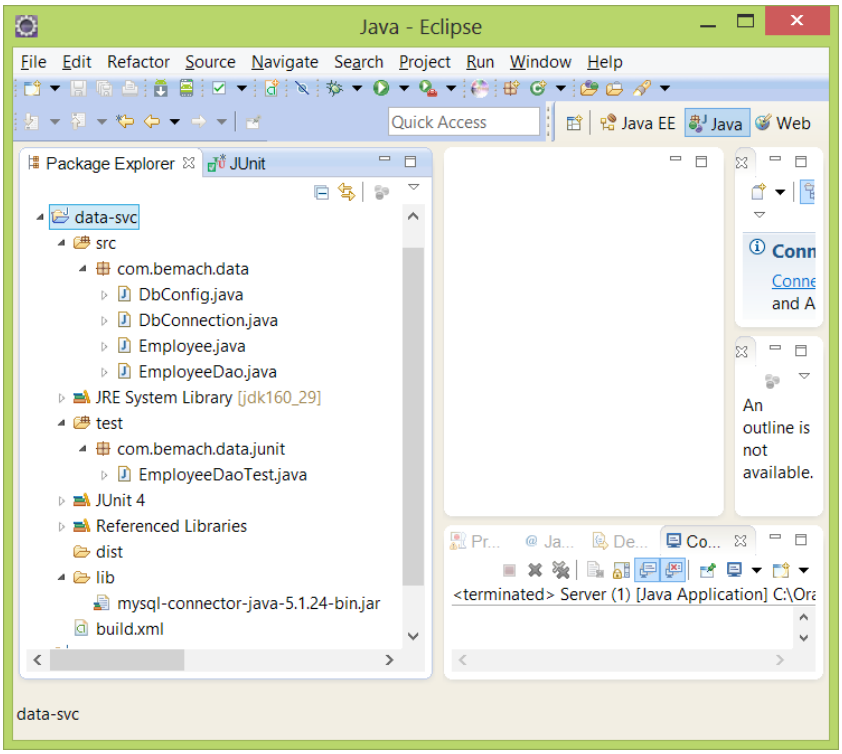


Figure 4-6. *Java Project: data-svc*

4.2.2.1 Import JDBC driver to the project

Following the instructions in section 7.2 to install MySQL and download an appropriate JDBC driver for MySQL database, the JDBC driver that is used for this application is mysql-connector-java-5.1.24-bin.jar.

Create a folder named 'lib' under the data-svc project by first selecting the project. Then, choose File → New → Folder. This folder will contain the JDBC driver library. Expand the project by clicking on the triangle to the left of the project name.

Now, import the JDBC driver that you have downloaded by clicking on the lib folder. Then, choose File → Import… The Select screen pops up as follows:

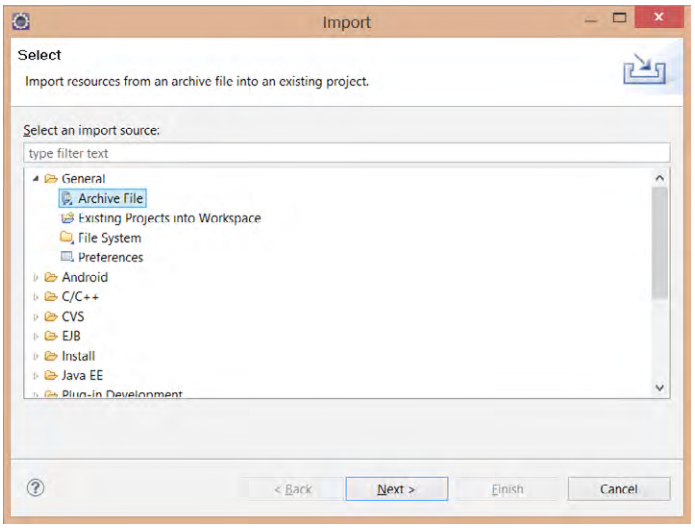


Figure 4-7. *Select import type*

Choose General → Archive File, and then click on the 'Next' button at the bottom of screen. An Archive file screen pops up as follows:

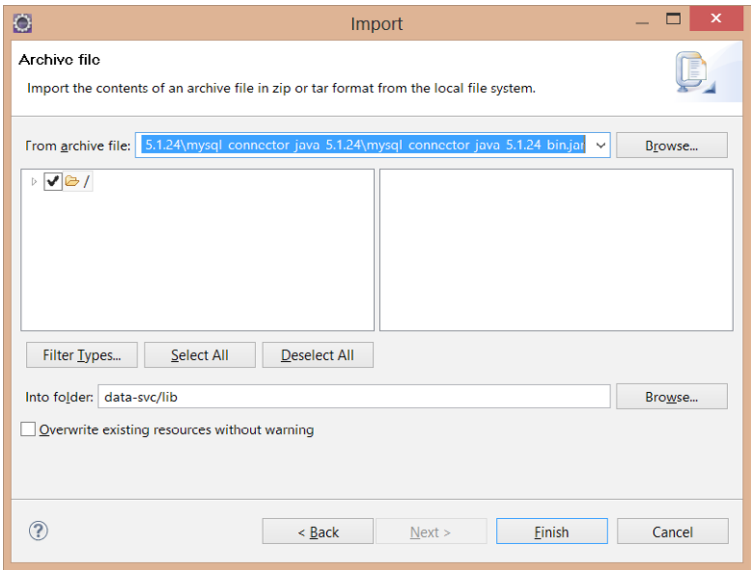


Figure 4-8. *Import Archive file screen*

If you know the location of the JDBC driver for MySQL database, enter the file name and location. Otherwise, use the 'Browse' button on the right and choose the file. Then, click the 'Finish' button on the bottom of the screen.

4.2.2.2 Reference to the library

Next, make sure the project has a reference to the MySQL JDBC driver library. First, choose the data-svc project. Then, select Project (menu) → Properties. The Properties for the data-svc screen will pop up as follows:

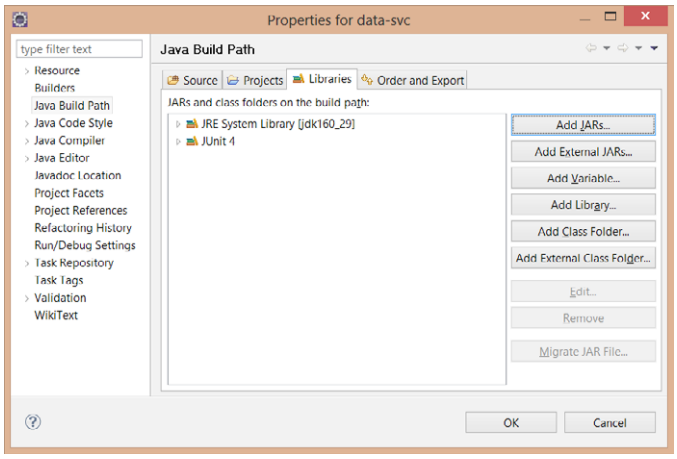


Figure 4-9. *Java Build Path*

On the left panel of the screen, choose Java Build Path. Select the Libraries tab from the top of the right panel. Click on the Add JARs… button. A JAR Selection screen will pop up as follows:

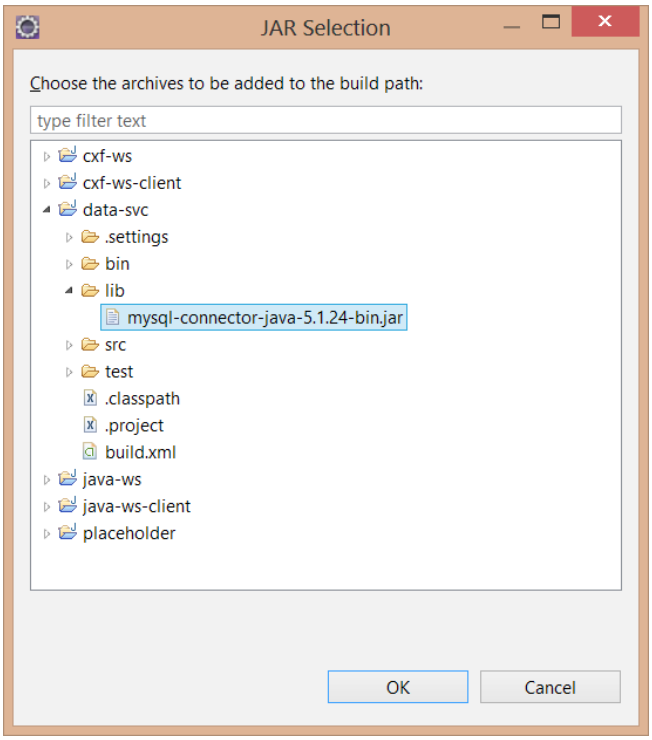


Figure 4-10. *JAR Selection screen*

Expand data-svc/lib folder. Select the JDBC driver. Then, click OK. Your Java Build Path screen should look like this:

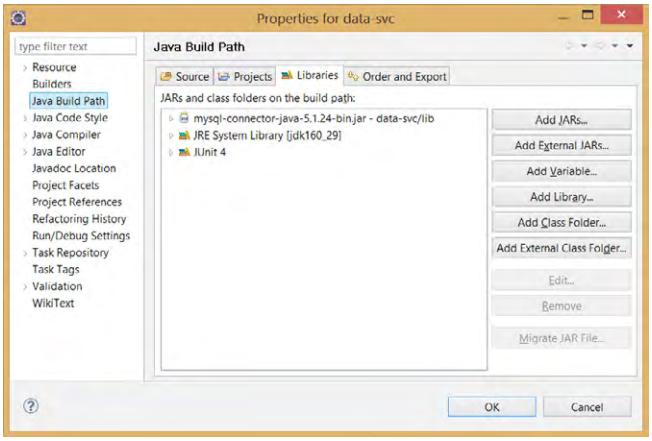


Figure 4-11. *Java Build Path*

Now, the coding can begin. In the following section, we create two classes – DbConnection.java and EmployeeDao.java.

4.2.2.3 DbConnection.java

In earlier sections, we created the DbConfig.java class to hold the configuration parameters for accessing the database. The next logical step is to create a class to manage all the JDBC connections for this application. Getting a database connection can also be accomplished using DataSource class; however, in this book, we use a basic method for obtaining a JDBC database connection.

Listing 4-8. DbConnection.java class

```
package com.bemach.data;
/**
 * 2013 (C) BEM, Inc., Fairfax, Virginia
 *
 * Unless required by applicable law or agreed to in writing,
 * software distributed is distributed on an
 * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
 * KIND, either express or implied.
 *
 */

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.logging.Level;
import java.util.logging.Logger;
```

```
public final class DbConnection {
   private static final Logger LOG = Logger.getLogger(DbConnection.class.getName());
   private static final String ERROR_MSG = "ERROR: ";
   private Connection conn = null;

   public Connection getConn() {
      return conn;
   }

   private DbConnection(String driverName, String subprot, String host,
            String port, String db, String uid, String psw) {
      LOG.info("Getting DB connection ...");

      try {
         Class.forName(driverName);
         String url = String.format("jdbc:%s://%s:%s/%s?user=%s&password=%s",
                              subprot, host, port, db, uid, psw);
         conn = DriverManager.getConnection(url);
      } catch (SQLException e) {
         LOG.log(Level.SEVERE,ERROR_MSG+e);
      } catch (ClassNotFoundException e) {
         LOG.log(Level.SEVERE,ERROR_MSG+e);
      }
   }

   public static DbConnection getInstance(String driverName, String subprot,
String host,
                           String port, String db, String uid, String psw) {
      return new DbConnection(driverName, subprot, host, port, db, uid, psw);
   }

   public void close() {
      try {
         if (conn != null) {
            conn.close();
            conn = null;
         }
      } catch (SQLException e) {
         LOG.log(Level.SEVERE,ERROR_MSG+e);
      }
   }
}
```

The DriverManager class helps create a JDBC connection in three ways.

`getConnection(String url)`

`getConnection(String url, Properties info)`

`getConnection(String url, String user, String password)`

In this example, the first method is used and the URL can be seen as follows:

```
jdbc:mysql://localhost:3306/employees?user=empl_1&password=pas
sword
```

The getConn() method requires all necessary database configuration parameters to connect to the database. When a JDBC connection is no longer needed, it must be explicitly closed by calling the closeConn() method. Accumulated open connections will strain the resources. In most JDBC drivers, closing a connection results in closing the Statement and Result sets that are associated with the connection.

4.2.2.4 EmployeeDao.java

This class provides basic access to the employees[2] table in the database. PreparedStatement is used to avoid potential SQL injection attack.

Our task is to develop a WS called EmployeeDataService that allows a client to create, read, update and delete a row from the employees table. For now, we are not concerned with security – we simply want to show how this can be done thorugh a bottom-up approach to create a Web Service.

First, we create a class that allows access to this table. This class is called `EmployeeDao` and allows four basic operations on a row of the employees table. An Employee class represents each employee from the Java coding. EmployeeDao uses basic Java Database Connectivity (JDBC) to create a database connection, issues an SQL statement, and processes the return. It is a basic JDBC application.

Listing 4-9. EmployeeDao.java class

```
package com.bemach.data;

/**
 * 2013 (C) BEM, Inc., Fairfax, Virginia
 *
 * Unless required by applicable law or agreed to in writing,
 * software distributed is distributed on an
 * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
 * KIND, either express or implied.
 *
 */

import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Timestamp;
import java.util.Calendar;
import java.util.logging.Logger;
```

```
/**
 * This class allows its application to perform the four (4) basic operations
 * of an Employee resource:
 * 1. Create
 * 2. Read
 * 3. Update
 * 4. Delete
 *
 * CRUD is classic in a sense that it is most like what an application does to
 * an authorized resource.
 *
 * Additional methods are:
 * getEmployeeByLastName
 * getEmplByFirstLastName
 *
 */
public class EmployeeDao {
   public static final Logger LOG = Logger.getLogger(EmployeeDao.class.getName());
   private DbConfig cfg = null;

   public EmployeeDao() {

   }

   /**
    * Constructor
    * @param cfg
    */
   public EmployeeDao (DbConfig cfg) {
      this.cfg = cfg;
      LOG.info("Constructing EmployeeDao ...");
   }

   /**
    * From a ResultSet returns an Employee record.
    *
    * @param rs
    * @return
    */
   protected Employee getEmpl(ResultSet rs) throws SQLException {
      Employee empl = new Employee();
      Calendar cal = Calendar.getInstance();
      empl.setEmplNo(rs.getInt("emp_no"));
      cal.setTimeInMillis(rs.getTimestamp("birth_date").getTime());
      empl.setBirthDate(cal);
      empl.setFirstName(rs.getString("first_name"));
      empl.setLastName(rs.getString("last_name"));
      empl.setGender(rs.getString("gender"));
      cal = Calendar.getInstance();
      cal.setTimeInMillis(rs.getTimestamp("hire_date").getTime());
      empl.setHireDate(cal);
      return empl;
   }
```

```
    /**
     * Create a new employee.
     *
     * @param empl
     * @return
     */
    public int createEmpl(Employee empl) throws SQLException {
        LOG.info("Create an employee");
        DbConnection dbConn = DbConnection.getInstance(cfg.getDriverName(),
                cfg.getSubprot(), cfg.getHost(),
                cfg.getPort(), cfg.getDb(),
                cfg.getUid(), cfg.getPsw());

        String sql = "SELECT MAX(EMP_NO) FROM EMPLOYEES";
        PreparedStatement stmt = null;
        ResultSet rs = null;
        try {
            stmt = dbConn.getConn().prepareStatement(sql);
            stmt.execute();
            rs = stmt.getResultSet();
            rs.next();
            int nextEmplNo = rs.getInt(1);
            stmt.close();
            rs.close();

            sql = "INSERT INTO EMPLOYEES (EMP_NO, BIRTH_DATE, FIRST_NAME, LAST_
NAME, GENDER, HIRE_DATE) " +
                    "VALUES (?,?,?,?,?,?)";
            stmt = dbConn.getConn().prepareStatement(sql);
            int idx = 1;
            stmt.setInt(idx++, ++nextEmplNo);
            Timestamp ts = new Timestamp(empl.getBirthDate().getTimeInMillis());
            stmt.setTimestamp(idx++, ts);
            stmt.setString(idx++, empl.getFirstName());
            stmt.setString(idx++, empl.getLastName());
            stmt.setString(idx++, empl.getGender());
            ts = new Timestamp(empl.getHireDate().getTimeInMillis());
            stmt.setTimestamp(idx++, ts);
            stmt.execute();
            return nextEmplNo;
        } finally {
            if (stmt != null) {
                stmt.close();
            }
            if (rs != null) {
                rs.close();
            }
            dbConn.close();
        }
    }
```

```
        /**
         * Update an employee record.
         *
         * @param empl
         * @return
         */
        public boolean updateEmpl(Employee empl) throws SQLException {
                LOG.info("Update an employee");
                DbConnection dbConn = DbConnection.getInstance(cfg.getDriverName(),
                                cfg.getSubprot(), cfg.getHost(),
                                cfg.getPort(), cfg.getDb(),
                                cfg.getUid(), cfg.getPsw());

                String sql = "UPDATE EMPLOYEES SET BIRTH_DATE=?, FIRST_NAME=?, LAST_NAME=?,
GENDER=?, HIRE_DATE=? " +
                                        "WHERE EMP_NO=?";
                PreparedStatement stmt = null;

                try {
                        stmt = dbConn.getConn().prepareStatement(sql);
                        int idx = 1;
                        Timestamp ts = new Timestamp(empl.getBirthDate().getTimeInMillis());
                        stmt.setTimestamp(idx++, ts);
                        stmt.setString(idx++, empl.getFirstName());
                        stmt.setString(idx++, empl.getLastName());
                        stmt.setString(idx++, empl.getGender());
                        ts = new Timestamp(empl.getHireDate().getTimeInMillis());
                        stmt.setTimestamp(idx++, ts);
                        stmt.setInt(idx++, (int)empl.getEmplNo());

                        stmt.execute();
                        return true;
                } finally {
                        if (stmt != null) {
                                stmt.close();
                        }
                        dbConn.close();
                }
        }

        /**
         * Delete an employee by Employee Number
         * @param emplNo
         * @return
         */
        public boolean deleteEmpl(int emplNo) throws SQLException {
                LOG.info("Delete an employee");
                DbConnection dbConn = DbConnection.getInstance(cfg.getDriverName(),
                                cfg.getSubprot(), cfg.getHost(),
                                cfg.getPort(), cfg.getDb(),
                                cfg.getUid(), cfg.getPsw());

                String sql = "DELETE FROM EMPLOYEES WHERE EMP_NO=?";
                PreparedStatement stmt = null;

                try {
                        stmt = dbConn.getConn().prepareStatement(sql);
                        stmt.setInt(1, emplNo);

                        stmt.execute();
                        return true;
                } finally {
                                if (stmt != null) {
                                        stmt.close();
                                }
                        dbConn.close();
                }
        }
```

```
        /**
         * Get an employee of a given unique employee number ..
         *
         * @param emplNo
         * @return
         */
        public Employee getEmpl (int emplNo) throws SQLException {
                LOG.info("Getting employee by Employee number: "+emplNo);
                DbConnection dbConn = DbConnection.getInstance(cfg.getDriverName(),
                                cfg.getSubprot(), cfg.getHost(),
                                cfg.getPort(), cfg.getDb(),
                                cfg.getUid(), cfg.getPsw());

                String sql = "SELECT * FROM EMPLOYEES WHERE EMP_NO=?";
                PreparedStatement stmt = null;
                ResultSet rs = null;
                try {
                        stmt = dbConn.getConn().prepareStatement(sql);
                        stmt.setInt(1, emplNo);
                        if (stmt.execute()) {
                                rs = stmt.getResultSet();
                                if (rs != null && rs.next()) {
                                        return getEmpl(rs);
                                }
                        }
                } finally {
                        if (stmt != null) {
                                stmt.close();
                        }
                        if (rs != null) {
                                rs.close();
                        }
                        dbConn.close();
                }
                return null;
        }

}
```

This class has four important methods that can create, read, update and delete an employee record from the employees table in the database. These methods will be reflected later through the four operations of the two Web Services called 'EmployeeDocData' and 'EmployeeRpcData'.

4.2.2.4.1 createEmpl(Employee empl)

This method receives a new employee record called 'empl' and inserts it into the employees table using a JDBC PreparedStatement class. This method assumes that the callers of this method have already validated the content of the employee record. Another approach is to include a validation method in this class and call from each of the four operations. Notice that all operations dealing with the database are done through a PreparedStatement in order to limit SQL inject attacks from the outside.

First, we get the largest employee number in order to create a new employee record with a unique primary key. This way of getting an employee number may encounter a concurrency problem when another application or process inserts another record at the same time; however, we ignore this condition here for the sake of simplicity.

Once a new employee number has been received, the method inserts the record into the database and a new employee number is returned to the caller; however, the final clause will first make sure that the database connection has been closed. This is one of the many techniques to ensure that resources are properly deallocated after the method completes its task.

4.2.2.4.2 getEmpl(emplNo)

This method retrieves an employee record from the employees table of the database. A unique employee number is a required input. If the record is found, it is returned to the caller. Otherwise, an exception is thrown. Regardless, at this point, the database connection is closed.

4.2.2.4.3 updateEmpl(empl)

This method updates a record with all values from the input record except the employee number. A Boolean value of true or false is returned after the processing is completed. If the record exists and the update completes successfully, a Boolean value of truth is returned. Otherwise, the method returns false.

4.2.2.4.4 deleteEmpl(emplNo)

Similarly to other methods of this class, this method opens a database connection then issues an SQL statement to complete the task. After a successful completion, an employee record will be removed from the table and a Boolean value of truth is returned. Otherwise, the method returns a value of false.

Overall, this class is relatively simple. It performs the most frequent operations on a resource stored in the database. This class is kept simple because our focus is on the creation of Web Services not database operations. When more complex business rules and multi-datasource data access activities are involved, the fundamental concept of Web Services remains the same. The main focus of Web Service is the interface – it must be robust and capable of evolving over time.

In the next section, we briefly discuss how to test the data access object so that we can ensure some basic quality assurance of the development team.

4.2.2.5 JUnit Test for Data Access Object

We provided a basic JUnit test for the EmployeeDao.java class. This class is called 'EmployeeDataTest.java'.

Listing 4-10. EmployeeDaoTest.java Class

```
package com.bemach.data.junit;

/**
 * 2013 (C) BEM, Inc., Fairfax, Virginia
 *
 * Unless required by applicable law or agreed to in writing,
 * software distributed is distributed on an
 * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
 * KIND, either express or implied.
 *
 */

import static org.junit.Assert.*;

import java.sql.SQLException;
import java.sql.Timestamp;
import java.util.Calendar;
import java.util.logging.Logger;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

import com.bemach.data.DbConfig;
import com.bemach.data.Employee;
import com.bemach.data.EmployeeDao;

public class EmployeeDaoTest {
   public static final Logger logger = Logger.getLogger(EmployeeDaoTest.class.
getName());

   /**
    * @throws java.lang.Exception
    */
   @BeforeClass
   public static void setUpBeforeClass() throws Exception {
   }

   /**
    * @throws java.lang.Exception
    */
   @AfterClass
   public static void tearDownAfterClass() throws Exception {
   }

   private EmployeeDao dao;
```

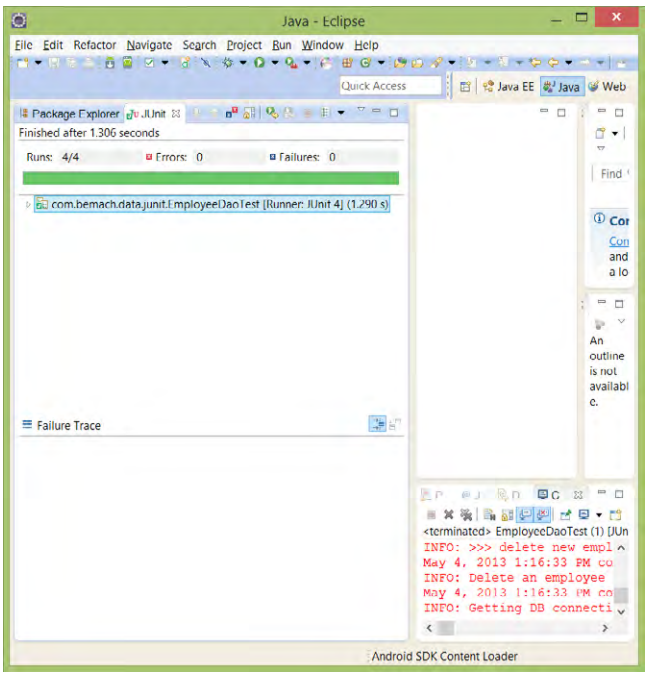
```
    /**
     * @throws java.lang.Exception
     */
    @Before
    public void setUp() throws Exception {
        logger.info("Testing employee dao class ...");
        DbConfig cfg = new DbConfig();
        cfg.setDriverName("com.mysql.jdbc.Driver");
        cfg.setHost("saintmonica");
        cfg.setPort("3306");
        cfg.setDb("employees");
        cfg.setUid("empl_1");
        cfg.setPsw("password");
        dao = new EmployeeDao(cfg);
    }

    /**
     * @throws java.lang.Exception
     */
    @After
    public void tearDown() throws Exception {
    }

    /**
     * Test method for {@link com.bemach.data.EmployeeDao#getEmpl(int)}.
     * @throws SQLException
     */
    @Test
    public void testGetEmplByEmplNo() throws SQLException {
        Employee empl = dao.getEmpl(10327);
        assertTrue("*** ERROR NULL ***", empl != null);
        logger.info("found "+empl.getFirstName()+"/"+empl.getLastName());
    }

    @Test
    public void testCRUDEmpl() throws SQLException {
        logger.info(">>> get empl");
        Employee empl = dao.getEmpl(10001);
        empl.setFirstName("Test_First");
        empl.setLastName("Test_Last");
        Timestamp ts = Timestamp.valueOf("1970-01-01 0:0:0.0");
        Calendar cal = Calendar.getInstance();
        cal.setTimeInMillis(ts.getTime());
        empl.setBirthDate(cal);
        ts = Timestamp.valueOf("1970-01-01 0:0:0.0");
        cal.setTimeInMillis(ts.getTime());
        empl.setHireDate(cal);
        empl.setGender("F");

        logger.info(">>> create empl");
        int newEmplNo = dao.createEmpl(empl);
        logger.info(">>> get new empl");
        Employee newEmpl = dao.getEmpl(newEmplNo);
```

```
        newEmpl.setGender("M");
        logger.info(">>> update new empl");
        dao.updateEmpl(newEmpl);
        logger.info(">>> get new empl again");
        newEmpl = dao.getEmpl(newEmplNo);
        printOutput(newEmpl);
        logger.info(">>> delete new empl");
        dao.deleteEmpl(newEmplNo);
    }

    private void printOutput(Employee empl) {
        StringBuffer sb = new StringBuffer();
        sb.append(", emplno=").append(empl.getEmplNo());
        sb.append(", fname=").append(empl.getFirstName());
        sb.append(", lname=").append(empl.getLastName());
        sb.append(", hire=").append(empl.getHireDate());
        sb.append(", birth=").append(empl.getBirthDate());
        sb.append(", gender=").append(empl.getGender());
        logger.info(sb.toString());
    }

}
```

After each operation, an employee record is formatted and displayed on the screen.

If all the tests are run successfully, the result should be displayed in green on the JUnit panel on the left-hand side of the Eclipse IDE:

Listing 4-11. JUnit test result



4.2.3 Package a Java Library

In order to run a build from a command line, JAVA_HOME and ANT_HOME variables need to be defined. Make sure to include $JAVA_HOME/bin or %JAVA_HOME%\bin in the PATH variable. JAVA_HOME should be pointed to the installed JDK. We've developed and tested with JDK 1.6. The Ant build script was of version 1.7.1.

This build.xml build script by default runs from the root of the project data-svc directory. The classes are stored in the bin directory, while the Java library data-svc.jar will be stored in the dist directory. A clean build command will remove both directories. Thus, two build commands should be used:

```
Ant dist (or simply ant)
Ant clean
```

Listing 4-12. build.xml for data-svc Java Project

```
<project name="data-svc" default="dist" basedir=".">
   <description>
      Data Services
   </description>

   <!- set global properties for this build ->
   <property environment="env" />
   <path id="classpath.base">
      <fileset dir="./lib" includes="**/*.jar" />
   </path>

   <path id="classpath.compile">
      <path refid="classpath.base" />
   </path>

   <target name="init">
      <mkdir dir="./bin" />
      <mkdir dir="./dist" />
   </target>

   <target name="compile" depends="init" description="compile the source „>
      <javac srcdir="./src" destdir="./bin" debug="true">
         <classpath refid="classpath.compile" />
      </javac>
   </target>

   <target name="dist" depends="compile" description="generate the distribution">
      <!- Create the distribution directory ->
      <jar jarfile="./dist/data-svc.jar" basedir="./bin" />

   </target>

   <target name="clean" description="clean up">
      <!- Delete the ${build} directory trees ->
      <delete dir="./dist" />
      <delete dir="./bin" />
   </target>

</project>
```

The output, a data-svc.jar file, is stored in the dist directory. The content of this JAR file should be as follows:

```
META-INF/
META-INF/MANIFEST.MF
com/
com/bemach/
com/bemach/data/
```

```
com/bemach/data/DbConfig.class
com/bemach/data/DbConnection.class
com/bemach/data/Employee.class
com/bemach/data/EmployeeDao.class
```

From the standpoint of business, EmployeeDao should be tested to ensure that it works at the level of the basic unit. All operations of the classes were thoroughly tested to ensure that the classes work.

4.2.4 Develop Java Classes for Web Services

To create a Java project under Eclipse IDE, please refer to Chapter 7. Import two libraries (i.e., data-svc.jar and MySQL JDBC driver) into the lib folder under java-ws project. Also, make sure to have these libraries in the Java Build Path. Refer to the previous section for instructions on how to make reference to the libraries for a Java project in Eclipse.

We develop a Web Service for employee with two different styles: document and RPC. First, we create a Java project called 'java-ws'. After we finish coding the required Java classes, the java-ws project should appear as follows:

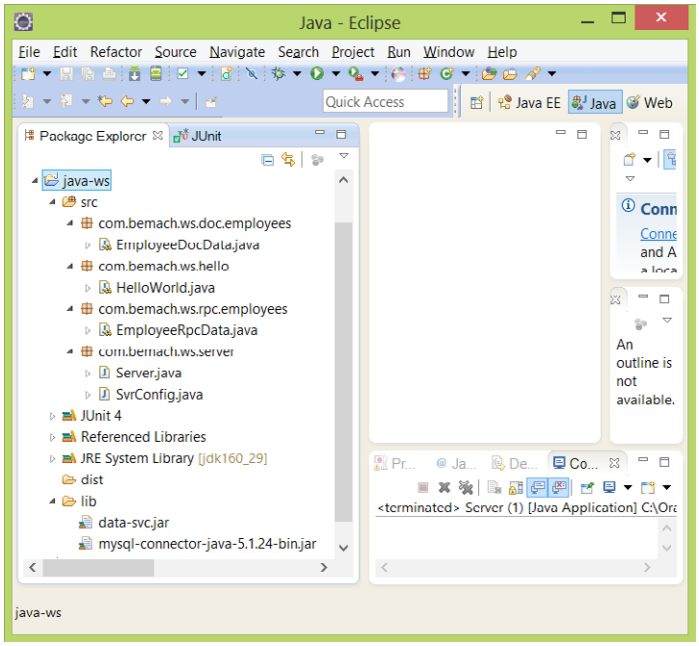


Figure 4-12. *java-ws Java Project*

The following sections describe the steps required to create Web Services in two main Java classes – EmployeeDocData.java and EmployeeRpcData.java

4.2.4.1 EmployeeDocData.java

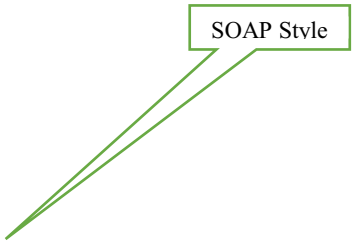
Writing Web Service in Java can be done by incorporating Java annotation into Java classes. These classes provide WS using SOAP. Java WS annotations that are used include the following:

- `@WebService:` indicates this class to implement a Web Service
- `@SOAPBinding:` specifies Web Service to bind to a SOAP protocol
- `@WebMethod:` exposes an operation as a Web method.
- `@WebParam:` mapps individual parameters to a WS message.

A document-style SOAP binding is used for this application.

Listing 4-13. EmployeeDocData.java Class

```
package com.bemach.ws.doc.employees;
/**
 * 2013 (C) BEM, Inc., Fairfax, Virginia
 *
 * Unless required by applicable law or agreed to in writing,
 * software distributed is distributed on an
 * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
 * KIND, either express or implied.
 *
 */


import java.util.logging.Logger;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.xml.soap.SOAPException;

import com.bemach.data.DbConfig;
import com.bemach.data.Employee;
import com.bemach.data.EmployeeDao;

@WebService
@SOAPBinding(style=SOAPBinding.Style.DOCUMENT)
public class EmployeeDocData  {
   private static final Logger LOG =
Logger.getLogger(EmployeeDocData.class.getName());
   private EmployeeDao dao = null;

   public EmployeeDocData (DbConfig cfg) {
      dao = new EmployeeDao(cfg);
   }

   @WebMethod
   public Employee getEmployee(@WebParam(name="emplNo")long emplNo) throws
SOAPException, Exception {
      LOG.info("Doc.readEmployee");
      Employee employee = dao.getEmpl((int)emplNo);
      if (employee == null) {
         throw new SOAPException ("No such employee!");
      }
      return employee;
   }

   @WebMethod
   public long createEmployee(@WebParam (name="employee")Employee employee) throws
Exception  {
      LOG.info("Doc.createEmployee");
      return dao.createEmpl(employee);
   }

   @WebMethod
   public boolean updateEmployee(@WebParam (name="employee")Employee employee)
throws Exception  {
      LOG.info("Doc.updateEmployee.");
      return dao.updateEmpl(employee);
   }

   @WebMethod
   public boolean deleteEmployee(@WebParam(name="emplNo")long emplNo) throws
Exception  {
      LOG.info("Doc.deleteEmployee.");
      return dao.deleteEmpl((int)emplNo);
   }
}
```



This class has four major operations (i.e., CRUD) on an employee record. Note that for each operation, the service creates an EmployeeDao object to call the matching operation. The call is then returned as the return of the operation of the service. Note that the marshalling of the return object is accomplished with the assistance of the JAXB component in Java.

These four operations are simple. Each method calls the corresponding method provided by EmployeeDao instance.

4.2.4.1.1 @WebService Annotation

@WebService annotation indicates that this class (or an interface) impelements a Web Service. This annotation has six (6) optional elements that can be used for a more detailed definition of a Web Service:

1. `endpointInterface:` the complete name of the service endpoint interface
2. `name:` the name of the `<portType>` element within the WSDL
3. `portName:` the name of the `<port>` element within the WSDL
4. `serviceName:` the name of the `<service>` element within the WSDL
5. `targetNamespace:` the targetNamespace attribute of the `<definition>` element of the WSDL
6. `wsdlLocation:` the content of the location attribute of the `<soap:address>` element

In this application, we did not include these optional elements. We will define the location of the WSDL when we create an `Endpoint within the server code (Server.java).`

4.2.4.1.2 @SOAPBinding Annotation

This annotation specifies how to map a Web Service onto the SOAP message protocol. These involve three optional elements:

1. `parameterStyle:` This can be either BARE or WRAPPED.
2. `style:` This can be either DOCUMENT or RPC
3. `use:` This can be either LITERAL or ENCODED.

In this sample application, we use DOCUMENT and RPC styles for two separate Web Services.

4.2.4.1.3 @WebMethod Annotation

This annotation customizes a method that is exposed as a WS operation. There are three (3) optional elements that can be used with this annotation:

1. `action`: name of an operation defined within the WSDL
2. `exclude`: excludes the method from being exposed as an operation of a Web Service
3. `operationName`: name of the operation.

4.2.4.1.4 @WebParam Annotation

Individual parameters of an operation can be named in the same way as the method. Use this annotation to change to different names within the WSDL. Optional parameters are:

1. `header`: if true, the parameter is extracted from the message header instead of from the message body.
2. `mode`: there are three basic modes – IN, OUT, and INOUT.
3. `name`: the parameter is mapped to name in XML element that represents the parameter. If DOCUMENT style is used, name is required.
4. `partName`: if RPC style is used, this is the name in the wsdl:part element.
5. `targetNamespace`: if DOCUMENT style is used, the parameter maps to a header.

4.2.4.2 EmployeeRpcData.java

This class implements the SOAP RPC style of Web Service. It is nearly identical to that of the document style with the exception of SOAPBinding annotation. This class is used to show the difference between the two styles in use today.

Listing 4-14. EmployeeRpcData.java Class

```
package com.bemach.ws.rpc.employees;
/**
 * 2013 (C) BEM, Inc., Fairfax, Virginia
 *
 * Unless required by applicable law or agreed to in writing,
 * software distributed is distributed on an
 * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
 * KIND, either express or implied.
 *
 */

import java.util.logging.Logger;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.xml.soap.SOAPException;

import com.bemach.data.DbConfig;
import com.bemach.data.Employee;
import com.bemach.data.EmployeeDao;

@WebService
@SOAPBinding(style=SOAPBinding.Style.RPC)
public class EmployeeRpcData {
   private static final Logger LOG = Logger.getLogger(EmployeeRpcData.class.
getName());
   private EmployeeDao dao = null;

   public EmployeeRpcData (DbConfig cfg) {
      dao = new EmployeeDao(cfg);
   }

   @WebMethod
   public Employee getEmployee(@WebParam(name="emplNo")long emplNo) throws
SOAPException, Exception {
      LOG.info("Rpc.readEmployee");
      Employee employee = getDao().getEmpl((int)emplNo);
      if (employee == null) {
         throw new SOAPException ("No such employee!");
      }
      return employee;
   }

   @WebMethod
   public long createEmployee(@WebParam (name="employee")Employee employee)
throws Exception {
      LOG.info("Rpc.createEmployee");
      return getDao().createEmpl(employee);
   }
```

SOAP Style

```
    @WebMethod
    public boolean updateEmployee(@WebParam (name="employee")Employee employee)
throws Exception {
        LOG.info("Rpc.updateEmployee.");
        return getDao().updateEmpl(employee);
    }

    @WebMethod
    public boolean deleteEmployee(@WebParam(name="emplNo")long emplNo) throws
Exception {
        LOG.info("Rpc.deleteEmployee.");
        return getDao().deleteEmpl((int)emplNo);
    }

    public EmployeeDao getDao() {
        return dao;
    }

    public void setDao(EmployeeDao dao) {
        this.dao = dao;
    }
}
```

4.2.5 Hosting Web Services

Web Services need to be hosted by a server that provides some basic HTTP service endpoints. Note that this type of server is rather simplistic in its implementation for the purpose of WS demonstration. A more industrial-strength application server, such as WebLogic, JBOSS, or WebSphere, is more appropriate for medium-sized to large business settings.

4.2.5.1 Server.java

This class implements a HTTP server to host multiple Web Services (e.g., HelloWorld, EmployeeDocDataService and EmployeeRpcDataService). Each WS is uniquely identified with a service endpoint.

- HelloWorld Web Service: http://localhost:9999/java-ws/hello?WSDL
- Employee Document Web Service: http://localhost:9999/doc/employees?wsdl
- Employee RPC Web Service: http://localhost:9999/rpc/employees?wsdl

Listing 4-15. Server.java Class

```
package com.bemach.ws.server;

/**
 * 2013 (C) BEM, Inc., Fairfax, Virginia
 *
 * Unless required by applicable law or agreed to in writing,
 * software distributed is distributed on an
 * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
 * KIND, either express or implied.
 *
 */

import java.util.logging.Logger;

import javax.xml.ws.Endpoint;
import javax.xml.ws.EndpointReference;

import com.bemach.data.DbConfig;
import com.bemach.ws.doc.employees.EmployeeDocData;
import com.bemach.ws.hello.HelloWorld;
import com.bemach.ws.rpc.employees.EmployeeRpcData;

/**
 *
 */
public final class Server {
   private static final Logger LOG = Logger.getLogger(Server.class.getName());
   private static final String MYSQL_DRIVER="com.mysql.jdbc.Driver";
   private static final String DB_HOST = "saintmonica";
   private static final String DB_PORT = "3306";
   private static final String DB_SID = "employees";
   private static final String DB_USER = "empl_1";
   private static final String DB_PSW = "password";
   private Server() {

   }

   protected static DbConfig getDbConfig() {
      DbConfig dbCfg = new DbConfig();
      dbCfg.setDriverName(MYSQL_DRIVER);
      dbCfg.setHost(DB_HOST);
      dbCfg.setPort(DB_PORT);
      dbCfg.setDb(DB_SID);
      dbCfg.setUid(DB_USER);
      dbCfg.setPsw(DB_PSW);
      return dbCfg;
   }
```

```
    private static final String HOST_NAME = "localhost";
    private static final String PORT_NO = "9999";
    private static final String HELLO_SVC_NAME = "java-ws/hello";
    private static final String RPC_EMPL_SVC_NAME = "rpc/employees";
    private static final String DOC_EMPL_SVC_NAME = "doc/employees";
    private static final String PROTOCOL = "http";

    protected static SvrConfig getSvrConfig() {
        SvrConfig svrCfg = new SvrConfig();
        svrCfg.setListenHostname(HOST_NAME);
        svrCfg.setListenPort(PORT_NO);
        svrCfg.setListenInterface(HELLO_SVC_NAME);
        svrCfg.setListenProtocol(PROTOCOL);
        return svrCfg;
    }

    protected static Endpoint publish(SvrConfig cfg, Object svc) {
        String url = String.format("%s://%s:%s/%s",
                  cfg.getListenProtocol(),
                  cfg.getListenHostname(),
                  cfg.getListenPort(),
                  cfg.getListenInterface());
           Endpoint ep = Endpoint.publish(url, svc);
           EndpointReference epr = ep.getEndpointReference();
           LOG.info("\nEndpoint Ref:\n"+epr.toString());
           return ep;
    }

    protected static void startHelloWorld() {
        SvrConfig cfg = getSvrConfig();
        cfg.setListenHostname(HOST_NAME);
        cfg.setListenInterface(HELLO_SVC_NAME);
        cfg.setListenPort(PORT_NO);
        cfg.setListenProtocol(PROTOCOL);

        HelloWorld hello = new HelloWorld();
           publish(cfg, hello);
           LOG.info("HelloWorld service started successfully ...");
    }

    protected static void startRpcEmployees() {
        SvrConfig svrCfg = getSvrConfig();
        svrCfg.setListenHostname(HOST_NAME);
        svrCfg.setListenInterface(RPC_EMPL_SVC_NAME);
        svrCfg.setListenPort(PORT_NO);
        svrCfg.setListenProtocol(PROTOCOL);
           DbConfig dbCfg = getDbConfig();
           svrCfg.setDbCfg(dbCfg);

        EmployeeRpcData rpcEmpl = new EmployeeRpcData(dbCfg);
           publish(svrCfg, rpcEmpl);
           LOG.info("Employees (RPC) service started successfully ...");
    }
```

```
    protected static void startDocEmployees() {
        SvrConfig svrCfg = getSvrConfig();
        svrCfg.setListenHostname(HOST_NAME);
        svrCfg.setListenInterface(DOC_EMPL_SVC_NAME);
        svrCfg.setListenPort(PORT_NO);
        svrCfg.setListenProtocol(PROTOCOL);
            DbConfig dbCfg = getDbConfig();
            svrCfg.setDbCfg(dbCfg);

        EmployeeDocData docEmpl = new EmployeeDocData(dbCfg);
            publish(svrCfg, docEmpl);

            LOG.info("Employees (Document) service started successfully ...");
    }

    /**
     * Start WS Server with multiple service endpoints...
     *
     * @param args
     */
    public static void main(String[] args) {
        startHelloWorld();
        startRpcEmployees();
        startDocEmployees();
     }
}
```

4.2.5.2 Package the Web Services

This Ant build script builds the java-ws.jar library and stores it in the dist directory. This build requires two Java libraries: data-svc.jar and mysql-connector-java-5.1.24-bin.jar.

Listing 4-16. build.xml for java-ws Java Project

```
<project name="java-ws" default="dist" basedir=".">
   <description>
      Web Service usign Java.
   </description>

   <!- set global properties for this build ->
   <property environment="env" />
   <path id="classpath.base">
      <fileset dir="./lib" includes="**/*.jar" />
   </path>

   <path id="classpath.compile">
      <path refid="classpath.base" />
   </path>

   <target name="init">
      <mkdir dir="./bin" />
      <mkdir dir="./dist" />
   </target>

   <target name="compile" depends="init" description="compile the source „>
      <javac srcdir="./src" destdir="./bin" debug="true">
         <classpath refid="classpath.compile" />
      </javac>
   </target>

   <target name="dist" depends="compile" description="generate the distribution">
      <!- Create the distribution directory ->
      <jar jarfile="./dist/java-ws.jar" basedir="./bin" />

   </target>

   <target name="clean" description="clean up">
      <!- Delete the ${build} directory trees ->
      <delete dir="./dist" />
      <delete dir="./bin" />
   </target>

</project>
```

The output of this build is a JAR file stored in the dist directory. The contents of this library consist of the following elements:

```
META-INF/
META-INF/MANIFEST.MF
com/
com/bemach/
com/bemach/ws/
```

```
com/bemach/ws/doc/
com/bemach/ws/doc/employees/
com/bemach/ws/hello/
com/bemach/ws/rpc/
com/bemach/ws/rpc/employees/
com/bemach/ws/server/
com/bemach/ws/doc/employees/EmployeeDocData.class
com/bemach/ws/hello/HelloWorld.class
com/bemach/ws/rpc/employees/EmployeeRpcData.class
com/bemach/ws/server/Server.class
com/bemach/ws/server/SvrConfig.class
```

4.3 Deploy Web Services

The server instance runs indefinitely. Use control-C to terminate the process. An alternative way to get the configuration parameters is to load them from a Java properties file. Note that, for Windows, the CLASSPATH separator is semi-colon (;) as opposed to colon (:) on UNIX.

```
java -cp ./dist/java-ws.jar;../data-svc/dist/data-svc.jar;./lib/mysql-
connector-java-5.1.24-bin.jar com.bemach.ws.server.Server
```

`mysql-connector-java-5.1.24-bin.jar` is a JDBC driver for MySQL database.

Next, we use SOAP to test the Web Services.

4.4 Check WSDL and XSD

We produce three services with three distinct service endpoints. After the server is running, we verify that these service endpoints are active and ready for service invocations. From a browser, we go to the URLs. The service endpoint for the HelloWorld example, http://localhost:9999/java-ws/hello?WSDL, was examined in earlier chapters. We visit the two `employees` service endpoints:

4.4.5.1 Document style

WSDL and XSD of the employees Web Service are shown in the following listings. A client application developer uses these WSDL documents to generate a Web Service stub for use inside their application.

Listing 4-17. WSDL of a DOCUMENT Style

```
<definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://employees.doc.ws.bemach.com/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
targetNamespace="http://employees.doc.ws.bemach.com/"
   name="EmployeeDocDataService">
   <types>
      <xsd:schema>
         <xsd:import namespace="http://employees.doc.ws.bemach.com/"
            schemaLocation="http://localhost:9999/doc/employees?xsd=1" />
      </xsd:schema>
      <xsd:schema>
         <xsd:import namespace="http://bemach.com"
            schemaLocation="http://localhost:9999/doc/employees?xsd=2" />
      </xsd:schema>
   </types>
   <message name="getEmployee">
      <part name="parameters" element="tns:getEmployee" />
   </message>
   <message name="getEmployeeResponse">
      <part name="parameters" element="tns:getEmployeeResponse" />
   </message>
   <message name="SOAPException">
      <part name="fault" element="tns:SOAPException" />
   </message>
   <message name="createEmployee">
      <part name="parameters" element="tns:createEmployee" />
   </message>
   <message name="createEmployeeResponse">
      <part name="parameters" element="tns:createEmployeeResponse" />
   </message>
   <message name="updateEmployee">
      <part name="parameters" element="tns:updateEmployee" />
   </message>
   <message name="updateEmployeeResponse">
      <part name="parameters" element="tns:updateEmployeeResponse" />
   </message>
   <message name="deleteEmployee">
      <part name="parameters" element="tns:deleteEmployee" />
   </message>
   <message name="deleteEmployeeResponse">
      <part name="parameters" element="tns:deleteEmployeeResponse" />
   </message>
   <portType name="EmployeeDocData">
      <operation name="getEmployee">
         <input message="tns:getEmployee" />
         <output message="tns:getEmployeeResponse" />
         <fault message="tns:SOAPException" name="SOAPException" />
      </operation>
      <operation name="createEmployee">
         <input message="tns:createEmployee" />
         <output message="tns:createEmployeeResponse" />
      </operation>
```

```
    <operation name="updateEmployee">
      <input message="tns:updateEmployee" />
      <output message="tns:updateEmployeeResponse" />
    </operation>
    <operation name="deleteEmployee">
      <input message="tns:deleteEmployee" />
      <output message="tns:deleteEmployeeResponse" />
    </operation>
  </portType>
  <binding name="EmployeeDocDataPortBinding" type="tns:EmployeeDocData">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
      style="document" />
    <operation name="getEmployee">
      <soap:operation soapAction="" />
      <input>
        <soap:body use="literal" />
      </input>
      <output>
        <soap:body use="literal" />
      </output>
      <fault name="SOAPException">
        <soap:fault name="SOAPException" use="literal" />
      </fault>
    </operation>
    <operation name="createEmployee">
      <soap:operation soapAction="" />
      <input>
        <soap:body use="literal" />
      </input>
      <output>
        <soap:body use="literal" />
      </output>
    </operation>
    <operation name="updateEmployee">
      <soap:operation soapAction="" />
      <input>
        <soap:body use="literal" />
      </input>
      <output>
        <soap:body use="literal" />
      </output>
    </operation>
    <operation name="deleteEmployee">
      <soap:operation soapAction="" />
      <input>
        <soap:body use="literal" />
      </input>
      <output>
        <soap:body use="literal" />
      </output>
    </operation>
  </binding>
```

```
    <service name="EmployeeDocDataService">
       <port name="EmployeeDocDataPort" binding="tns:EmployeeDocDataPortBinding">
          <soap:address location="http://localhost:9999/doc/employees" />
       </port>
    </service>
</definitions>
```

Consider the operation getEmployee (highlighted). This operation has one input and one output element. These elements are defined in the message area above. These messages are getEmployee and getEmployeeResponse, which are of tns:getEmployee and tns:getEmployeeResponse types, respectively. The types are defined in the schema located at http://localhost:9999/doc/employees?xsd=1. See highlighted area.

URL for associated schema:

Listing 4-18. Schema (XSD) of a Web Service

```
<xs:schema xmlns:tns="http://employees.doc.ws.bemach.com/"
   xmlns:ns1="http://bemach.com" xmlns:xs="http://www.w3.org/2001/XMLSchema"
   version="1.0" targetNamespace="http://employees.doc.ws.bemach.com/">
   <xs:import namespace="http://bemach.com"
      schemaLocation="http://localhost:9999/doc/employees?xsd=2" />
   <xs:element name="SOAPException" type="tns:SOAPException" />
   <xs:element name="createEmployee" type="tns:createEmployee" />
   <xs:element name="createEmployeeResponse" type="tns:createEmployeeResponse" />
   <xs:element name="deleteEmployee" type="tns:deleteEmployee" />
   <xs:element name="deleteEmployeeResponse" type="tns:deleteEmployeeResponse" />
   <xs:element name="getEmployee" type="tns:getEmployee" />
   <xs:element name="getEmployeeResponse" type="tns:getEmployeeResponse" />
   <xs:element name="updateEmployee" type="tns:updateEmployee" />
   <xs:element name="updateEmployeeResponse" type="tns:updateEmployeeResponse" />
   <xs:complexType name="deleteEmployee">
      <xs:sequence>
         <xs:element name="emplNo" type="xs:long" />
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="deleteEmployeeResponse">
      <xs:sequence>
         <xs:element name="return" type="xs:boolean" />
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="createEmployee">
      <xs:sequence>
         <xs:element name="employee" type="tns:employee" minOccurs="0" />
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="employee">
      <xs:sequence>
         <xs:element name="emplNo" type="xs:long" />
         <xs:element name="firstName" type="xs:string" />
         <xs:element name="lastName" type="xs:string" />
         <xs:element name="birthDate" type="xs:dateTime" />
         <xs:element name="gender" type="xs:string" />
         <xs:element name="hireDate" type="xs:dateTime" />
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="createEmployeeResponse">
      <xs:sequence>
         <xs:element name="return" type="xs:long" />
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="getEmployee">
      <xs:sequence>
         <xs:element name="emplNo" type="xs:long" />
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="getEmployeeResponse">
      <xs:sequence>
         <xs:element name="return" type="tns:employee" minOccurs="0" />
      </xs:sequence>
   </xs:complexType>
```

```
   <xs:complexType name="SOAPException">
      <xs:sequence>
         <xs:element name="message" type="xs:string" minOccurs="0" />
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="updateEmployee">
      <xs:sequence>
         <xs:element name="employee" type="tns:employee" minOccurs="0" />
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="updateEmployeeResponse">
      <xs:sequence>
         <xs:element name="return" type="xs:boolean" />
      </xs:sequence>
   </xs:complexType>
</xs:schema>
```

4.4.5.2 RPC Style

The difference between the WSDLs of RPC and Document styles can be difficult to detect; however, XSDs are visibly different. All data types for the document style are defined using XML schema, while all the simple data types (e.g., integer, long, string) are defined within the WSDL.

Listing 4-19. WSDL of a RPC Style

```
<definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://employees.rpc.ws.bemach.com/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
targetNamespace="http://employees.rpc.ws.bemach.com/"
   name="EmployeeRpcDataService">
   <types>
      <xsd:schema>
         <xsd:import namespace="http://employees.rpc.ws.bemach.com/"
            schemaLocation="http://localhost:9999/rpc/employees?xsd=1" />
      </xsd:schema>
      <xsd:schema>
         <xsd:import namespace="http://bemach.com"
            schemaLocation="http://localhost:9999/rpc/employees?xsd=2" />
      </xsd:schema>
   </types>
   <message name="getEmployee">
      <part name="emplNo" type="xsd:long" />
   </message>
   <message name="getEmployeeResponse">
      <part name="return" type="tns:employee" />
   </message>
   <message name="SOAPException">
      <part name="fault" element="tns:SOAPException" />
   </message>
   <message name="createEmployee">
      <part name="employee" type="tns:employee" />
   </message>
   <message name="createEmployeeResponse">
      <part name="return" type="xsd:long" />
   </message>
   <message name="updateEmployee">
      <part name="employee" type="tns:employee" />
   </message>
   <message name="updateEmployeeResponse">
      <part name="return" type="xsd:boolean" />
   </message>
   <message name="deleteEmployee">
      <part name="emplNo" type="xsd:long" />
   </message>
   <message name="deleteEmployeeResponse">
      <part name="return" type="xsd:boolean" />
   </message>
   <portType name="EmployeeRpcData">
      <operation name="getEmployee">
         <input message="tns:getEmployee" />
         <output message="tns:getEmployeeResponse" />
         <fault message="tns:SOAPException" name="SOAPException" />
      </operation>
      <operation name="createEmployee">
         <input message="tns:createEmployee" />
         <output message="tns:createEmployeeResponse" />
      </operation>
```

```
    <operation name="updateEmployee">
       <input message="tns:updateEmployee" />
       <output message="tns:updateEmployeeResponse" />
    </operation>
    <operation name="deleteEmployee">
       <input message="tns:deleteEmployee" />
       <output message="tns:deleteEmployeeResponse" />
    </operation>
 </portType>
 <binding name="EmployeeRpcDataPortBinding" type="tns:EmployeeRpcData">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
       style="rpc" />
    <operation name="getEmployee">
       <soap:operation soapAction="" />
       <input>
          <soap:body use="literal" namespace="http://employees.rpc.ws.bemach.com/" />
       </input>
       <output>
          <soap:body use="literal" namespace="http://employees.rpc.ws.bemach.com/" />
       </output>
       <fault name="SOAPException">
          <soap:fault name="SOAPException" use="literal" />
       </fault>
    </operation>
    <operation name="createEmployee">
       <soap:operation soapAction="" />
       <input>
          <soap:body use="literal" namespace="http://employees.rpc.ws.bemach.com/" />
       </input>
       <output>
          <soap:body use="literal" namespace="http://employees.rpc.ws.bemach.com/" />
       </output>
    </operation>
    <operation name="updateEmployee">
       <soap:operation soapAction="" />
       <input>
          <soap:body use="literal" namespace="http://employees.rpc.ws.bemach.com/" />
       </input>
       <output>
          <soap:body use="literal" namespace="http://employees.rpc.ws.bemach.com/" />
       </output>
    </operation>
    <operation name="deleteEmployee">
       <soap:operation soapAction="" />
       <input>
          <soap:body use="literal" namespace="http://employees.rpc.ws.bemach.com/" />
       </input>
       <output>
          <soap:body use="literal" namespace="http://employees.rpc.ws.bemach.com/" />
       </output>
    </operation>
 </binding>
```

Style

```
   <service name="EmployeeRpcDataService">
      <port name="EmployeeRpcDataPort" binding="tns:EmployeeRpcDataPortBinding">
         <soap:address location="http://localhost:9999/rpc/employees" />
      </port>
   </service>
</definitions>
```

Unlike the document style, the XSD documents for the RPC style are kept simple. Most of the basic data types are defined inside the WSDL instead of in the XSD.

Listing 4-20. XSD of a Web Service (RPC)

```
<xs:schema xmlns:tns="http://employees.rpc.ws.bemach.com/"
   xmlns:xs="http://www.w3.org/2001/XMLSchema" version="1.0"
   targetNamespace="http://employees.rpc.ws.bemach.com/">
   <xs:element name="SOAPException" type="tns:SOAPException" />
   <xs:complexType name="employee">
      <xs:sequence>
         <xs:element name="emplNo" type="xs:long" />
         <xs:element name="firstName" type="xs:string" />
         <xs:element name="lastName" type="xs:string" />
         <xs:element name="birthDate" type="xs:dateTime" />
         <xs:element name="gender" type="xs:string" />
         <xs:element name="hireDate" type="xs:dateTime" />
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="SOAPException">
      <xs:sequence>
         <xs:element name="message" type="xs:string" minOccurs="0" />
      </xs:sequence>
   </xs:complexType>
</xs:schema>
```

Listing 4-21. An Additional XSD of a Web Service

```
<xs:schema xmlns:ns1="http://employees.rpc.ws.bemach.com/"
   xmlns:xs="http://www.w3.org/2001/XMLSchema" version="1.0"
   targetNamespace="http://bemach.com">
   <xs:import namespace="http://employees.rpc.ws.bemach.com/"
      schemaLocation="http://localhost:9999/rpc/employees?xsd=1" />
   <xs:element name="EmployeeService" type="ns1:employee" />
</xs:schema>
```

4.5 Test Web Services with SOAPUI

First, create a SOAPUI project for each of the WS endpoints. Then, run the operation of each Web Service.

4.5.1 SOAPUI projects

Web Service can be tested using SOAPUI, which is an open source cross-platform functional testing tool that can be used to test Web Services. Like Eclipse, SOAPUI is organized into projects. Each project usually manages one service endpoint. Each service endpoint contains one or more operations that can be called from a client machine. In order to test a Web Service, all you really need is a service endpoint URL provided by your service provider.

The following figure shows how to create a SOAPUI test project for the employees data service with document style at this service endpoint: http://localhost:9999/doc/employees?WSDL.

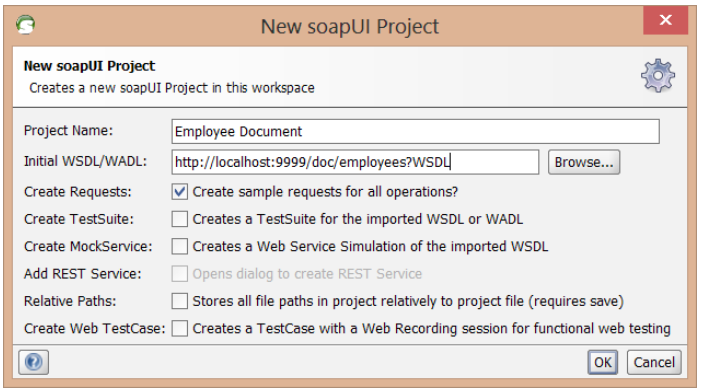


Figure 4-13. *Create a SOAPUI Project*

Once the project has been created, a set of operations will appear, as shown in the following figures. Note that you can now start testing these WS operations. In our example, four operations are visible: createEmployee, deleteEmployee, getEmployee, and updateEmployee.

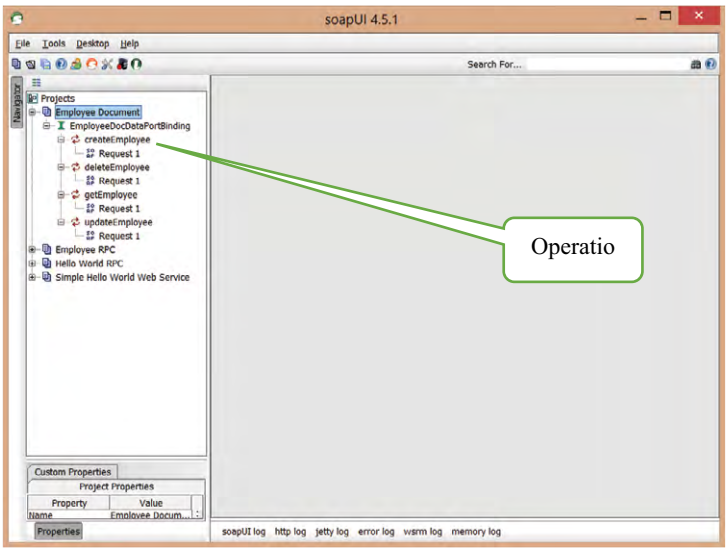


Figure 4-14. *List of Oerations of a Web Service*

Double-clicking on Request1 of the createEmployee operation will cause a multi-pane window to be displayed. You can fill in the parameters and run the test by clicking on the green triangle to the left panel.

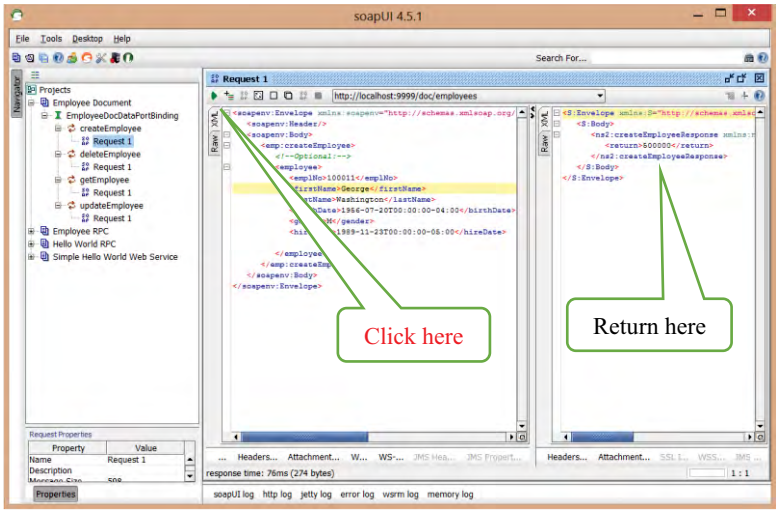


Figure 4-15. *Execute a SOAP Operations*

Other operations of the service can be tested in the same way.

Web Service with an RPC style can be tested in a similar way. The only difference is that when you create a SOAPUI project, you provide a different service endpoint (URL). A SOAPUI project can be created for EmployeeData service with RPC style as follows:

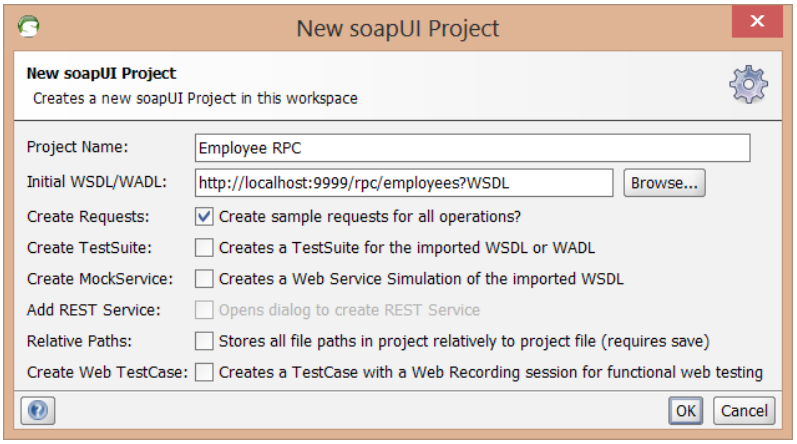


Figure 4-16. *Create a new SOAPUI Project*

4.6 Develope a Web Service Consumer

Developing a WS consumer (or client) involves three major activities: creating a client stub, creating a client code that uses the client stub to call service operations, and running the client. These activities are described as follows.

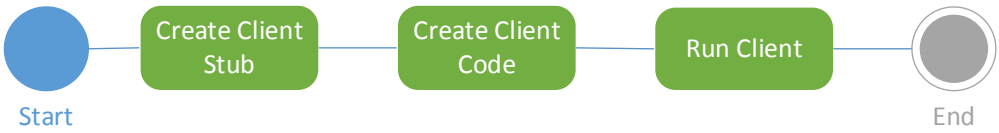


Figure 4-17. *Activities for Creating a Web Service Client*

4.6.1 Creating Client Stub with wsimport

Writing a SOAP client with SAAJ can be complex and time-consuming. See section 2.4 for details. Instead, we will use the wsimport tool to generate WS artifacts (stubs).

First, create a Java Project under Eclipse IDE and call it 'java-ws-client'.

1. At the command prompt, go to the Java Project for Eclipse called 'java-ws-client'.
2. Create a folder called 'generated'.
3. To generate WS stubs, run the following commands:

```
wsimport -d . http://localhost:9999/doc/employees?WSDL
wsimport -d . http://localhost:9999/rpc/employees?WSDL
wsimport -d . http://localhost:9999/java-ws/hello?WSDL
```

4. To create a Java library, run the following command:

```
jar -cvf ../lib/java-ws-generated.jar *
```

5. To verify the content of the created jar, run this command:

```
jar -tf ../lib/java-ws-generated.jar *
```

The content of the library should appears as follows:

```
META-INF/
META-INF/MANIFEST.MF
com/
com/bemach/
com/bemach/ObjectFactory.class
com/bemach/ws/
com/bemach/ws/doc/
com/bemach/ws/doc/employees/
com/bemach/ws/doc/employees/CreateEmployee.class
```

```
com/bemach/ws/doc/employees/CreateEmployeeResponse.class
com/bemach/ws/doc/employees/DeleteEmployee.class
com/bemach/ws/doc/employees/DeleteEmployeeResponse.class
com/bemach/ws/doc/employees/Employee.class
com/bemach/ws/doc/employees/EmployeeDocData.class
com/bemach/ws/doc/employees/EmployeeDocDataService.class
com/bemach/ws/doc/employees/GetEmployee.class
com/bemach/ws/doc/employees/GetEmployeeResponse.class
com/bemach/ws/doc/employees/ObjectFactory.class
com/bemach/ws/doc/employees/package-info.class
com/bemach/ws/doc/employees/SOAPException.class
com/bemach/ws/doc/employees/SOAPException_Exception.class
com/bemach/ws/doc/employees/UpdateEmployee.class
com/bemach/ws/doc/employees/UpdateEmployeeResponse.class
com/bemach/ws/hello/
com/bemach/ws/hello/HelloWorld.class
com/bemach/ws/hello/HelloWorldService.class
com/bemach/ws/hello/ObjectFactory.class
com/bemach/ws/hello/package-info.class
com/bemach/ws/hello/Say.class
com/bemach/ws/hello/SayResponse.class
com/bemach/ws/rpc/
com/bemach/ws/rpc/employees/
com/bemach/ws/rpc/employees/Employee.class
com/bemach/ws/rpc/employees/EmployeeRpcData.class
com/bemach/ws/rpc/employees/EmployeeRpcDataService.class
com/bemach/ws/rpc/employees/ObjectFactory.class
com/bemach/ws/rpc/employees/package-info.class
com/bemach/ws/rpc/employees/SOAPException.class
com/bemach/ws/rpc/employees/SOAPException_Exception.class
```

These commands generate java binary code that can become part of a client program that calls Web Services. Next, we create a Java library that contains the generated code: java-ws-generated.jar. This library should be included as part of a library set for the Eclipse IDE. It is also a part of the Java CLASSPATH during execution.

4.6.2 Create Client Code

We present two types of client code – document style and RPC style. Both are commonly used in WS programming today; however, document style is preferred.

After the coding has been completed, the java-ws-client project should look like this:

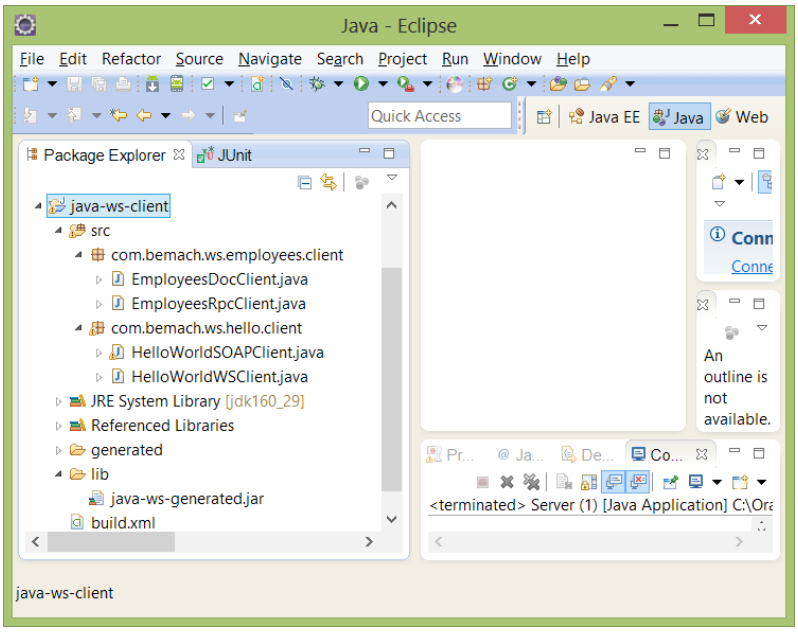


Figure 4-18. *Screenshot of java-ws-client Java Project*

4.6.2.1 EmployeesDocClient.java

This client code uses the generated client stub for making WS calls to the remote server. One important class is the QName, where we create a qualified name that contains the targetNamespace and the name attributes of the definitions element of the WSDL. The next important class is the URL where we create a service endpoint as the location attribute of the soap:address element of the WSDL. From these two classes, we can then create a service which we map onto the set of operations that the service provides. Mapping the port onto the EmployeeDocData is specified as a type attribute of the binding element within the WSDL. Once we get the port, we can call the operations as we did with the local method invocation in Java.

Listing 4-22. EmployeesDocClient.java Class

```
package com.bemach.ws.employees.client;
/**
 * 2013 (C) BEM, Inc., Fairfax, Virginia
 *
 * Unless required by applicable law or agreed to in writing,
 * software distributed is distributed on an
 * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
 * KIND, either express or implied.
 *
 */

import java.net.MalformedURLException;
import java.net.URL;
import java.util.logging.Logger;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;

import com.bemach.ws.doc.employees.Employee;
import com.bemach.ws.doc.employees.EmployeeDocData;
import com.bemach.ws.doc.employees.SOAPException_Exception;
import com.bemach.ws.rpc.employees.EmployeeRpcDataService;

/**
 * This code relies on ws client generated code using wsimport program:
 * wsimport -d . http://localhost:9999/doc/employees?WSDL
 * wsimport -d . http://localhost:9999/rpc/employees?WSDL
 * wsimport -d . http://localhost:9999/ch-1/HelloWorld?WSDL
 *
 * jar cvf ../ws-ch-1-generated.jar *
 *
 */
```

```
public class EmployeesDocClient {
   private static final Logger LOG = Logger.getLogger(EmployeesDocClient.
class.getName());
   private EmployeeDocData emplDs = null;

   public EmployeesDocClient(String urlStr, String targetNs, String name)
         throws MalformedURLException {
      LOG.info("Constructor ...");
      QName qName = new QName(targetNs, name);
      URL url = new URL(urlStr);
      Service service = EmployeeRpcDataService.create(url, qName);
      emplDs = service.getPort(EmployeeDocData.class);
   }

   public Employee get(long id) throws SOAPException_Exception {
      return emplDs.getEmployee(id);
   }

   public long create(Employee empl) {
      return emplDs.createEmployee(empl);
   }

   public boolean delete(long id) {
      return emplDs.deleteEmployee(id);
   }

   public boolean update(Employee empl) {
      return emplDs.updateEmployee(empl);
   }

   /**
    * @param args
    * @throws MalformedURLException
    * @throws SOAPException_Exception
    */
   public static void main(String[] args)
         throws MalformedURLException, SOAPException_Exception {
      LOG.info("Calling Employee (Document) data service ... ");
      String targetNameSpace = "http://employees.doc.ws.bemach.com/";
      String name = "EmployeeDocDataService";
      String urlStr = String.format("http://localhost:%s/doc/
employees",args[0]);

      EmployeesDocClient cli = new EmployeesDocClient(urlStr, targetNam-
eSpace, name);

      long oldEmplNo = Integer.valueOf(args[1]);
      Employee empl = cli.get(oldEmplNo);
      LOG.info("last="+empl.getLastName());
      LOG.info("hire="+empl.getHireDate());
      LOG.info("last="+empl.getLastName());
      LOG.info("first="+empl.getFirstName());
```

```
        empl.setFirstName("Silvester");
        empl.setLastName("Johnny");
        long newEmplNo = cli.create(empl);
        LOG.info("emplNo="+newEmplNo);

        Employee newEmpl = cli.get(newEmplNo);

        newEmpl.setLastName("New-name");
        newEmpl.setEmplNo(newEmplNo);
        boolean status = cli.update(newEmpl);
        LOG.info("update:"+status);
        LOG.info("last="+newEmpl.getLastName());
        LOG.info("first="+newEmpl.getFirstName());

        status = cli.delete(newEmplNo);
        LOG.info("deleteEmployee:"+status);
        LOG.info("Exit!");
    }
}
```

4.6.2.2 EmployeesRpcClient.java

This class is nearly identical to that of the document-style client code. Thus, any difference between the two styles of client code is nearly impossible to notice at this level. The significant difference is in the coding within the SOAP engine on the client side.

Listing 4-23. EmployeesRpcClient.java Class

```
package com.bemach.ws.employees.client;
/**
 * 2013 (C) BEM, Inc., Fairfax, Virginia
 *
 * Unless required by applicable law or agreed to in writing,
 * software distributed is distributed on an
 * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
 * KIND, either express or implied.
 *
 */

import java.net.MalformedURLException;
import java.net.URL;
import java.util.logging.Logger;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;

import com.bemach.ws.rpc.employees.Employee;
import com.bemach.ws.rpc.employees.EmployeeRpcData;
import com.bemach.ws.rpc.employees.EmployeeRpcDataService;
import com.bemach.ws.rpc.employees.SOAPException_Exception;
```

```
/**
 * This code relies on ws client generated code using wsimport program:
 * wsimport -d . http://localhost:9999/rpc/Employees?WSDL
 * jar cvf ../ws-ch-1-generated.jar *
 *
 */
public class EmployeesRpcClient {
   private static final Logger LOG = Logger.getLogger(EmployeesRpcClient.
class.getName());

   private EmployeeRpcData emplDs = null;

   public EmployeesRpcClient(String urlStr, String targetNs, String name)
throws MalformedURLException {
      LOG.info("Constructor ...");
      QName qName = new QName(targetNs, name);
      URL url = new URL(urlStr);
      Service service = EmployeeRpcDataService.create(url, qName);
      emplDs = service.getPort(EmployeeRpcData.class);
   }

   public Employee get(long id) throws SOAPException_Exception {
      return emplDs.getEmployee(id);
   }

   public long create(Employee empl) {
      return emplDs.createEmployee(empl);
   }

   public boolean delete(long id) {
      return emplDs.deleteEmployee(id);
   }

   public boolean update(Employee empl) {
      return emplDs.updateEmployee(empl);
   }

   /**
    * @param args
    * @throws MalformedURLException
    * @throws SOAPException_Exception
    */
   public static void main(String[] args)
         throws MalformedURLException, SOAPException_Exception {
      LOG.info("Calling Employee (RPC) data service ... ");

      String targetNameSpace = "http://employees.rpc.ws.bemach.com/";
      String name = "EmployeeRpcDataService";
      String urlStr = String.format("http://localhost:%s/rpc/employees",args[0]);

      EmployeesRpcClient cli = new EmployeesRpcClient(urlStr, targetNameSpace, name);
```

```
        long oldEmplNo = Integer.valueOf(args[1]);
        Employee empl = cli.get(oldEmplNo);

        LOG.info("last="+empl.getLastName());
        LOG.info("hire="+empl.getHireDate());
        LOG.info("last="+empl.getLastName());
        LOG.info("first="+empl.getFirstName());

        empl.setFirstName("Silvester");
        empl.setLastName("Johnny");
        long newEmplNo = cli.create(empl);
        LOG.info("emplNo="+newEmplNo);

        Employee newEmpl = cli.get(newEmplNo);

        newEmpl.setLastName("New-name");
        newEmpl.setEmplNo(newEmplNo);
        boolean status = cli.update(newEmpl);
        LOG.info("update:"+status);
        LOG.info("last="+newEmpl.getLastName());
        LOG.info("first="+newEmpl.getFirstName());

        status = cli.delete(newEmplNo);
        LOG.info("deleteEmployee:"+status);
        LOG.info("Exit!");
    }
}
```

4.6.3 Run the Client Application

With each of the client codes, we implemented one of the main methods for unit-testing purposes. Each in this class can run as a standalone Java application. To run these applications, the following command is used:

```
java -cp java-ws-client.jar;./lib/java-ws-generated.jar com.bemach.
ws.employees.client.EmployeesDocClient
```

or

```
java -cp java-ws-client.jar;./lib/ws-ch-1-generated.jar com.bemach.
ws.employees.client.EmployeesRpcClient
```

5 Apache CXF and Tomcat Server

Objectives

After completing this chapter, you should be able to:

1. Develop a Java Web Application
2. Package a Web Application
3. Deploy a Web Application to Apache Tomcat 7 server

Deploying a Web Service using JDK alone is not adequate for large and complex systems. In this chapter, we will learn to develop and deploy Web Services using Apache CXF Web Services and Apache Tomcat server 7. Tomcat is an open source software program that implements Java Servlet and Java Server Page (JSP) technologies.

Apache CXF can be downloaded from Apache's website http://cxf.apache.org. Follow the download and installation instructions for your machine. The CXF WS project includes many of its Java libraries. The list of required libraries is shown in this section.

5.1 Configuration Parameters

The default port used for the Tomcat server is 8080. We keep this default for the Tomcat server that hosts the sample application we developed in the previous chapter. In fact, we use all of the default values that come with the Tomcat server.

5.2 Apache Tomcat Server

To run a Tomcat 7 server, go to the apache-tomcat-7.0.12/bin directory and run startup.cmd or startup.sh.

5.3 Develop CXF Web Service

Developing a Web Service using the Apache CXF software package is similar to that of the reference implementation that comes with the JDK.

5.3.1 Class Diagram

Consider the following updated class diagram. In this diagram, we added two classes – EmployeeDataIf.java and EmployeeData.java. The latter implements the interface of the former.

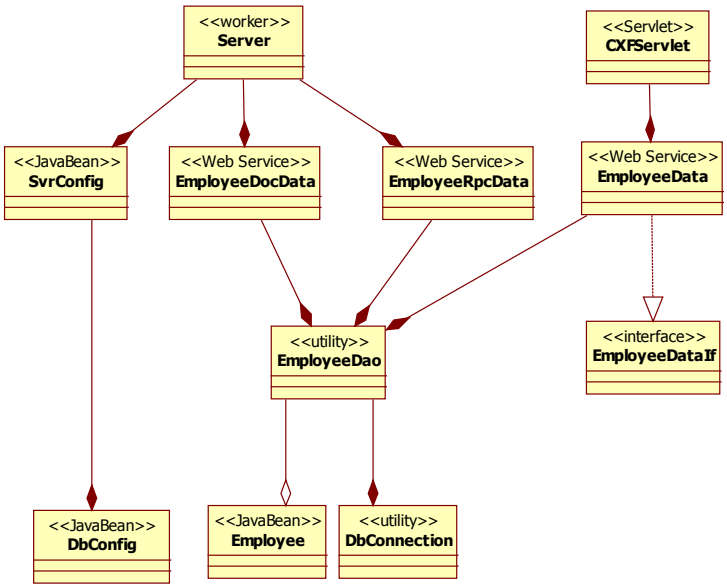


Figure 5-1. *Class Diagram for a CXF Web Service*

5.3.2 Deployment Diagram

We will deploy the Java Web Application onto the Tomcat 7 server.

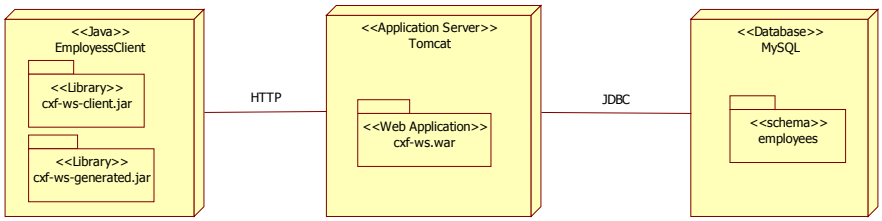


Figure 5-2. *Deployment Diagram for a CXF Web Service*

Developing and deploying Web Services involves the following major activities:

1. Create a dynamic Web application.
2. Create Web Services in Java.
3. Package the WAR file.
4. Deploy the WAR file to the Tomcat 7 server.

Figure 5-3. *Activities for Creating a Web Service Application*

5.3.3 Create a Dynamic Web Application

From Eclipse, create a new dynamic Web Project. The steps are as follows:

- Choose File → New → Other … → Web → Dynamic Web Project
- Choose all default values and name the project 'cxf-ws'.
- Choose cxf-ws project from the left panel, choose File → New → Folder. Name new folder lib.
- Now, we need the following Java libraries:

```
- mysql-connector-java-5.1.24-bin.jar: This contains the JDBC
  driver for MySQL database.
- java-ws.jar: This contains the main logic for accessing
  the database.
- data-svc.jar: This contains the data access classes.
```

- Import these files (from previous projects and from the Apache CXF distribution) into the lib folder we created earlier:

```
mysql-connector-java-5.1.24-bin.jar
ws-ch-1.jar
aopalliance-1.0.jar
asm-3.3.jar
axis.jar
commons-dbcp-1.4.jar
commons-discovery-0.2.jar
commons-logging-1.1.1.jar
commons-logging.jar
cxf-2.4.0.jar
geronimo-activation_1.1_spec-1.1.jar
geronimo-annotation_1.0_spec-1.1.1.jar
geronimo-javamail_1.4_spec-1.7.1.jar
geronimo-servlet_3.0_spec-1.0.jar
geronimo-ws-metadata_2.0_spec-1.1.3.jar
jaxb-api-2.2.1.jar
jaxb-impl-2.2.1.1.jar
jaxrpc.jar
neethi-3.0.0.jar
saaj-api-1.3.jar
saaj-impl-1.3.2.jar
spring-aop-3.0.5.RELEASE.jar
spring-asm-3.0.5.RELEASE.jar
spring-beans-3.0.5.RELEASE.jar
spring-context-3.0.5.RELEASE.jar
spring-core-3.0.5.RELEASE.jar
spring-expression-3.0.5.RELEASE.jar
spring-web-3.0.5.RELEASE.jar
stax2-api-3.1.1.jar
woodstox-core-asl-4.1.1.jar
wsdl4j-1.6.2.jar
wsdl4j.jar
xml-resolver-1.2.jar
xmlschema-core-2.0.jar
```

After we have finished coding Java classes for cxf-ws, the project should look like this:

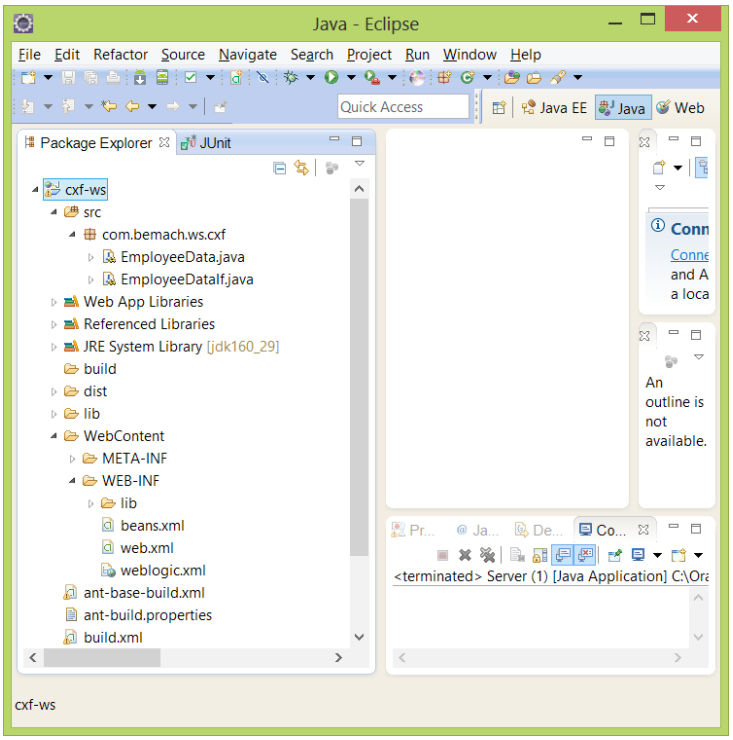


Figure 5-4. *Screenshot of a Dynamic Web Project*

5.3.4 Create Web Service

From the `cxf-ws/Java Resources/src` directory, create a Java package called '`com.bemach.ws.cxf`'. From this package, create one Java interface and one Java class.

5.3.4.1 EmployeeDataIf.java

We introduced another approach to create a Web Service using Java interface. This gives us a cleaner way to specify the service interface prior to implementing the Web Service. All WS annotations that were normally used for a Java class are used in a similar way.

Listing 5-1. EmployeeDataIf.java Class with Web Service Annotations

```
package com.bemach.ws.cxf;
/**
 * 2013 (C) BEM, Inc., Fairfax, Virginia
 *
 * Unless required by applicable law or agreed to in writing,
 * software distributed is distributed on an
 * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
 * KIND, either express or implied.
 *
 */

import java.sql.SQLException;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.xml.soap.SOAPException;

import com.bemach.data.Employee;

@WebService
@SOAPBinding(style=SOAPBinding.Style.DOCUMENT)
public interface EmployeeDataIf {
   @WebMethod
   Employee getEmployee(@WebParam(name="emplNo")long emplNo) throws
SOAPException, SQLException;

   @WebMethod
   long createEmployee(@WebParam (name="employee")Employee employee) throws
SOAPException, SQLException;

   @WebMethod
   boolean updateEmployee(@WebParam (name="employee")Employee employee) throws
SOAPException, SQLException;

   @WebMethod
   boolean deleteEmployee(@WebParam(name="emplNo")long emplNo) throws
SOAPException, SQLException;
}
```

The service will contain four basic operations:

1. The getEmployee method, which returns an employee of a given employee number. If not found, an exception is thrown;
2. The createEmployee method, which returns an employee number if created successfully. If failed, the method returns -1;
3. The updateEmployee method, which returns a Boolean value of true if successful, otherwise false; and
4. The deleteEmployee method, which returns a Boolean value of true if successful, otherwise false.

5.3.4.2 EmployeeData.java

This class is a wrapper class of the EmployeeDao.java class that we have seen earlier. It implements the WS interface EmployeeDataIf.

Listing 5-2. EmployeeData.java: An implementation of a Web Service Interface

```
package com.bemach.ws.cxf;
/**
 * 2013 (C) BEM, Inc., Fairfax, Virginia
 *
 * Unless required by applicable law or agreed to in writing,
 * software distributed is distributed on an
 * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
 * KIND, either express or implied.
 *
 */

import java.sql.SQLException;
import java.util.logging.Logger;

import javax.jws.WebService;
import javax.xml.soap.SOAPException;

import com.bemach.data.DbConfig;
import com.bemach.data.Employee;
import com.bemach.data.EmployeeDao;

@WebService(endpointInterface="com.bemach.ws.cxf.EmployeeDataIf")
public class EmployeeData implements EmployeeDataIf {
   private static final Logger LOG = Logger.getLogger(EmployeeData.class.getName());
   private static DbConfig cfg = new DbConfig();

   public static DbConfig getCfg() {
      return cfg;
   }

   public static void setCfg(DbConfig cfg) {
      EmployeeData.cfg = cfg;
   }

   public EmployeeData(DbConfig cfg) {
      EmployeeData.cfg = cfg;
   }

   public EmployeeData() {
   }

   @Override
   public Employee getEmployee(long emplNo) throws SOAPException, SQLException
{
      LOG.info("read employee");
      EmployeeDao dao = new EmployeeDao(cfg);
      Employee employee;
      employee = dao.getEmpl((int) emplNo);

      if (employee == null) {
         throw new SOAPException("GetEmployee: No such employee!");
      }
```

```
        return employee;
    }

    @Override
    public long createEmployee(Employee employee) throws SOAPException, SQLException {
        LOG.info("create employee");
        EmployeeDao dao = new EmployeeDao(cfg);
        return dao.createEmpl(employee);
    }

    @Override
    public boolean updateEmployee(Employee employee) throws SOAPException,
SQLException {
        LOG.info("update employee.");
        EmployeeDao dao = new EmployeeDao(cfg);
        return dao.updateEmpl(employee);
    }

    @Override
    public boolean deleteEmployee(long emplNo) throws SOAPException, SQLException {
        LOG.info("delete employee.");
        EmployeeDao dao = new EmployeeDao(cfg);
        return dao.deleteEmpl((int)emplNo);
    }
}
```

A default configuration is used. The EmployeeData constructor can be used to modify the configuration cfg object if necessary.

5.3.5 Package a WAR

The two important files – beans.xml and web.xml – are stored in the WEB-INF directory of the WAR file. These files assist the Tomcat Server and CXF to implement the Web Services that perform the actual work. Remember, we implement the sample Web Service using a Web application to be deployed on a Web application server (Tomcat or others).

5.3.5.1 Create web.xml

Listing 5-3. Content of web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
   <context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>WEB-INF/beans.xml</param-value>
   </context-param>

   <listener>
      <listener-class>
         org.springframework.web.context.ContextLoaderListener
      </listener-class>
   </listener>

   <servlet>
      <servlet-name>CXFServlet</servlet-name>
      <display-name>CXF Servlet</display-name>
      <servlet-class>
         org.apache.cxf.transport.servlet.CXFServlet
      </servlet-class>
      <load-on-startup>1</load-on-startup>
   </servlet>

   <servlet-mapping>
      <servlet-name>CXFServlet</servlet-name>
      <url-pattern>/*</url-pattern>
   </servlet-mapping>
</web-app>
```

5.3.5.2 Create beans.xml

Listing 5-4. Content of beans.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:jaxws="http://cxf.apache.org/jaxws"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">

       <import resource="classpath:META-INF/cxf/cxf.xml" />
       <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml" />
       <import resource="classpath:META-INF/cxf/cxf-servlet.xml" />

       <jaxws:endpoint id="com.bemach.ws.cxf.EmployeeDataIf"
implementor="com.bemach.ws.cxf.EmployeeData"address="/employees" />
</beans>
```

5.3.5.3 Build Web Application (WAR)

This build script creates a Java Web application (WAR file) packed into a file called 'cxf-ws.war'. The application is stored in the dist directory.

Listing 5-5. Content of build.xml for cxf-ws Dynamic Web Project

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project name="cxf-ws" basedir="." default="dist">

   <property environment="env" />
   <path id="classpath.base">
      <fileset dir="./lib" includes="**/*.jar" />
   </path>

   <path id="classpath.compile">
      <path refid="classpath.base" />
   </path>

   <target name="init">
      <mkdir dir="./bin" />
      <mkdir dir="./dist" />
   </target>

   <target name="compile" depends="init" description="compile the source ">
      <javac srcdir="./src" destdir="./bin" debug="true">
         <classpath refid="classpath.compile" />
      </javac>
   </target>

   <target name="copylibs" depends="compile">
      <echo message="copying libraries ..."/>
      <copy todir="WebContent/WEB-INF/lib">
         <fileset dir="./lib">
            <include name="*.jar"/>
         </fileset>
      </copy>
   </target>

   <target name="dist" depends="buildwar">
      <echo message="Building ..."/>
   </target>

   <target name="buildwar" depends="copylibs">
      <echo message="building war ...."/>
      <tstamp/>
      <manifest file="MANIFEST.MF">
            <attribute name="Built-By" value="Kiet T. Tran"/>
         <attribute name="Build-Version" value="1"/>
         <attribute name="Build-Subversion" value="0"/>
         <attribute name="Built-Date" value="April 28, 2013"/>
      </manifest>
```

```
      <war destfile="./dist/cxf-ws.war" webxml="WebContent/WEB-INF/web.xml"
update="true"
            manifest="MANIFEST.MF">
         <classes dir="./bin" />
         <fileset dir="WebContent">
            <exclude name="WEB-INF/web.xml" />
         </fileset>
      </war>
      <delete file="MANIFEST.MF"/>
   </target>

   <target name="clean">
      <echo message="cleaning ...."/>
      <delete dir="./dist"/>
      <delete dir="./bin"/>
   </target>
</project>
```

5.4 Deploy the Service

When you install Tomcat 7, the Tomcat server home directory has the following directories:

- bin
- conf
- lib
- logs
- temp
- webapps
- work

We are most interested in the bin and the webapps directories for deploying and publishing our Web Services. To run Tomcat, go to the bin directory and run startup.cmd (WINDOWS) or startup.sh (UNIX).

To deploy the Web application that contains the sample Web Services, copy the cxf-web.war to the webapps directory. You need not restart the Tomcat server each time you deploy the Web application.

5.5 Testing services with SOAPUI

The testing of the new Web Service that is running on the Apache Tomcat server is similar to that of the Java WS Endpoint deployment described earlier. The Web Service endpoint is:

http://localhost:8080/cxf-ws/employees?WSDL

5.5.1 Check WSDL

Listing 5-6. A WSDL of a CXF Web Service Application

```
<wsdl:definitions xmlns:ns1="http://schemas.xmlsoap.org/soap/http"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://cxf.ws.bemach.com/"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   name="EmployeeDataService" targetNamespace="http://cxf.ws.bemach.com/">
   <wsdl:types>
      <xs:schema xmlns:ns1="http://bemach.com" xmlns:tns="http://cxf.ws.bemach.com/"
         xmlns:xs="http://www.w3.org/2001/XMLSchema"
attributeFormDefault="unqualified"
         elementFormDefault="unqualified"
targetNamespace="http://cxf.ws.bemach.com/">
         <xs:import namespace="http://bemach.com" />
         <xs:element name="createEmployee" type="tns:createEmployee" />
         <xs:element name="createEmployeeResponse" type="tns:createEmployeeResponse"
/>
         <xs:element name="deleteEmployee" type="tns:deleteEmployee" />
         <xs:element name="deleteEmployeeResponse" type="tns:deleteEmployeeResponse"
/>
         <xs:element name="getEmployee" type="tns:getEmployee" />
         <xs:element name="getEmployeeResponse" type="tns:getEmployeeResponse" />
         <xs:element name="updateEmployee" type="tns:updateEmployee" />
         <xs:element name="updateEmployeeResponse" type="tns:updateEmployeeResponse"
/>
         <xs:complexType name="createEmployee">
            <xs:sequence>
               <xs:element minOccurs="0" name="employee" type="tns:employee" />
            </xs:sequence>
         </xs:complexType>
         <xs:complexType name="employee">
            <xs:sequence>
               <xs:element name="emplNo" type="xs:long" />
               <xs:element minOccurs="0" name="firstName" type="xs:string" />
               <xs:element minOccurs="0" name="lastName" type="xs:string" />
               <xs:element minOccurs="0" name="birthDate" type="xs:dateTime" />
               <xs:element minOccurs="0" name="gender" type="xs:string" />
               <xs:element minOccurs="0" name="hireDate" type="xs:dateTime" />
            </xs:sequence>
         </xs:complexType>
         <xs:complexType name="createEmployeeResponse">
            <xs:sequence>
               <xs:element name="return" type="xs:long" />
            </xs:sequence>
         </xs:complexType>
         <xs:complexType name="deleteEmployee">
            <xs:sequence>
               <xs:element name="emplNo" type="xs:long" />
            </xs:sequence>
         </xs:complexType>
```

```
        <xs:complexType name="deleteEmployeeResponse">
           <xs:sequence>
              <xs:element name="return" type="xs:boolean" />
           </xs:sequence>
        </xs:complexType>
        <xs:complexType name="getEmployee">
           <xs:sequence>
              <xs:element name="emplNo" type="xs:long" />
           </xs:sequence>
        </xs:complexType>
        <xs:complexType name="getEmployeeResponse">
           <xs:sequence>
              <xs:element minOccurs="0" name="return" type="tns:employee" />
           </xs:sequence>
        </xs:complexType>
        <xs:complexType name="updateEmployee">
           <xs:sequence>
              <xs:element minOccurs="0" name="employee" type="tns:employee" />
           </xs:sequence>
        </xs:complexType>
        <xs:complexType name="updateEmployeeResponse">
           <xs:sequence>
              <xs:element name="return" type="xs:boolean" />
           </xs:sequence>
```

```
         </xs:complexType>
         <xs:complexType name="SOAPException">
            <xs:sequence />
         </xs:complexType>
         <xs:element name="SOAPException" type="tns:SOAPException" />
      </xs:schema>
      <xs:schema xmlns:ns1="http://cxf.ws.bemach.com/"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
         targetNamespace="http://bemach.com" version="1.0">
         <xs:import namespace="http://cxf.ws.bemach.com/" />
         <xs:element name="EmployeeService" type="ns1:employee" />
      </xs:schema>
   </wsdl:types>
   <wsdl:message name="createEmployeeResponse">
      <wsdl:part element="tns:createEmployeeResponse" name="parameters"></wsdl:part>
   </wsdl:message>
   <wsdl:message name="updateEmployee">
      <wsdl:part element="tns:updateEmployee" name="parameters"></wsdl:part>
   </wsdl:message>
   <wsdl:message name="getEmployeeResponse">
      <wsdl:part element="tns:getEmployeeResponse" name="parameters"></wsdl:part>
   </wsdl:message>
   <wsdl:message name="SOAPException">
      <wsdl:part element="tns:SOAPException" name="SOAPException"></wsdl:part>
   </wsdl:message>
   <wsdl:message name="updateEmployeeResponse">
      <wsdl:part element="tns:updateEmployeeResponse" name="parameters"></wsdl:part>
   </wsdl:message>
   <wsdl:message name="deleteEmployeeResponse">
      <wsdl:part element="tns:deleteEmployeeResponse" name="parameters"></
wsdl:part>
   </wsdl:message>
   <wsdl:message name="getEmployee">
      <wsdl:part element="tns:getEmployee" name="parameters"></wsdl:part>
   </wsdl:message>
   <wsdl:message name="createEmployee">
      <wsdl:part element="tns:createEmployee" name="parameters"></wsdl:part>
   </wsdl:message>
   <wsdl:message name="deleteEmployee">
      <wsdl:part element="tns:deleteEmployee" name="parameters"></wsdl:part>
   </wsdl:message>
   <wsdl:portType name="EmployeeDataIf">
      <wsdl:operation name="createEmployee">
         <wsdl:input message="tns:createEmployee"
name="createEmployee"></wsdl:input>
         <wsdl:output message="tns:createEmployeeResponse"
name="createEmployeeResponse"></wsdl:output>
      </wsdl:operation>
      <wsdl:operation name="deleteEmployee">
         <wsdl:input message="tns:deleteEmployee"
name="deleteEmployee"></wsdl:input>
         <wsdl:output message="tns:deleteEmployeeResponse"
```

```
name="deleteEmployeeResponse"></wsdl:output>
      </wsdl:operation>
      <wsdl:operation name="getEmployee">
         <wsdl:input message="tns:getEmployee"
name="getEmployee"></wsdl:input>
         <wsdl:output message="tns:getEmployeeResponse"
name="getEmployeeResponse"></wsdl:output>
         <wsdl:fault message="tns:SOAPException" name="SOAPException"></wsdl:fault>
      </wsdl:operation>
      <wsdl:operation name="updateEmployee">
         <wsdl:input message="tns:updateEmployee"
name="updateEmployee"></wsdl:input>
         <wsdl:output message="tns:updateEmployeeResponse"
name="updateEmployeeResponse"></wsdl:output>
      </wsdl:operation>
   </wsdl:portType>
   <wsdl:binding name="EmployeeDataServiceSoapBinding" type="tns:EmployeeDataIf">
      <soap:binding style="document"
         transport="http://schemas.xmlsoap.org/soap/http" />
      <wsdl:operation name="createEmployee">
         <soap:operation soapAction="" style="document" />
         <wsdl:input name="createEmployee">
            <soap:body use="literal" />
         </wsdl:input>
         <wsdl:output name="createEmployeeResponse">
            <soap:body use="literal" />
         </wsdl:output>
      </wsdl:operation>
      <wsdl:operation name="deleteEmployee">
         <soap:operation soapAction="" style="document" />
         <wsdl:input name="deleteEmployee">
            <soap:body use="literal" />
         </wsdl:input>
         <wsdl:output name="deleteEmployeeResponse">
            <soap:body use="literal" />
         </wsdl:output>
      </wsdl:operation>
      <wsdl:operation name="getEmployee">
         <soap:operation soapAction="" style="document" />
         <wsdl:input name="getEmployee">
            <soap:body use="literal" />
         </wsdl:input>
         <wsdl:output name="getEmployeeResponse">
            <soap:body use="literal" />
         </wsdl:output>
         <wsdl:fault name="SOAPException">
            <soap:fault name="SOAPException" use="literal" />
         </wsdl:fault>
      </wsdl:operation>
      <wsdl:operation name="updateEmployee">
         <soap:operation soapAction="" style="document" />
         <wsdl:input name="updateEmployee">
            <soap:body use="literal" />
```

```
         </wsdl:input>
         <wsdl:output name="updateEmployeeResponse">
            <soap:body use="literal" />
         </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>
   <wsdl:service name="EmployeeDataService">
      <wsdl:port binding="tns:EmployeeDataServiceSoapBinding"
         name="EmployeeDataPort">
         <soap:address location="http://localhost:8080/cxf-ws/employees" />
      </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```

5.5.2 SOAPUI project

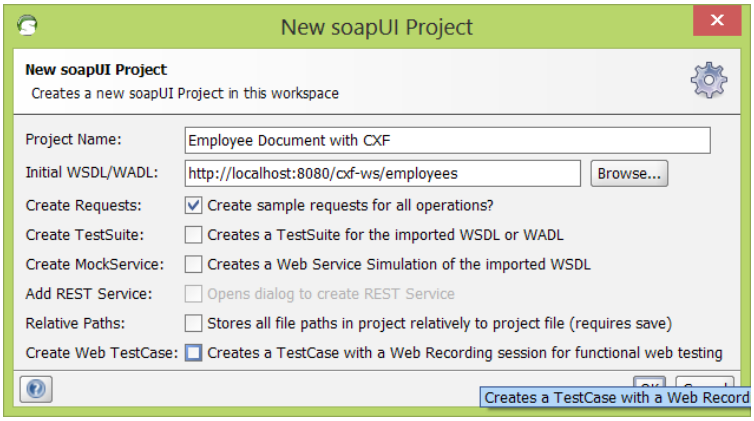


Figure 5-5. *Create a New SOAPUI Project*

The figure below shows a result of a call to the getEmployee operation.

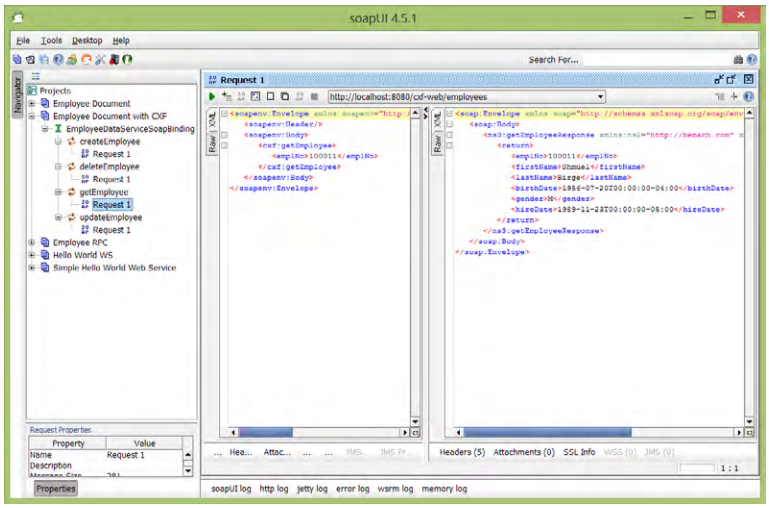


Figure 5-6. *Executing a Web Service Operation*

5.6 Develop a Web Service Consumer

5.6.1 Create Client Stub with wsimport

First, create a Java Project under Eclipse IDE and call it 'cxf-ws-client'.

1. At the command prompt, go to the Java Project for Eclipse called 'cxf-ws-client'.
2. Create a folder called 'generated'.
3. To generate Web Service stubs, run the following command:

```
wsimport -d . http://localhost:8080/cxf-ws/employees?WSDL
```

4. To create a Java library, run the following command:

```
jar -cvf ../lib/cxf-ws-generated.jar *
```

5. Now, verify the content of the created jar:

```
jar -tf ../lib/cxf-ws-generated.jar *
```

The content of the library should look like this:

```
META-INF/
META-INF/MANIFEST.MF
com/
com/bemach/
com/bemach/ObjectFactory.class
com/bemach/ws/
com/bemach/ws/cxf/
com/bemach/ws/cxf/Employee.class
com/bemach/ws/cxf/EmployeeDataIf.class
com/bemach/ws/cxf/EmployeeDataService.class
com/bemach/ws/cxf/ObjectFactory.class
com/bemach/ws/cxf/package-info.class
com/bemach/ws/cxf/SOAPException.class
com/bemach/ws/cxf/SOAPException_Exception.class
```

5.6.2 Create Client Code

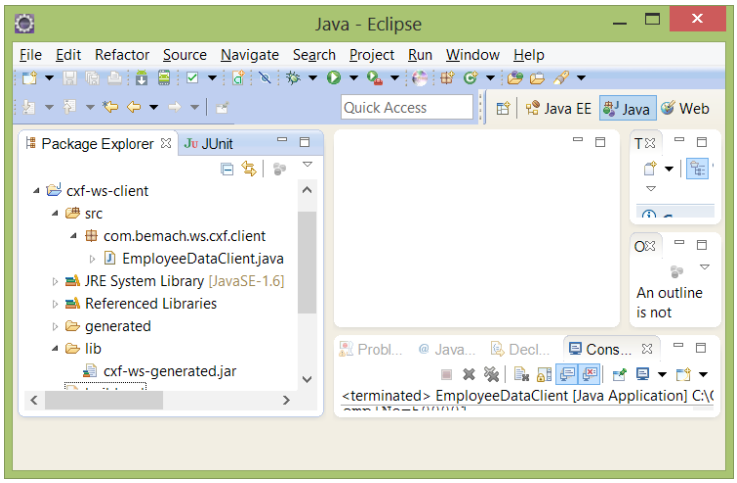


Figure 5-7. *Screenshot of cxf-ws-client Java Project*

5.6.2.1 EmployeeDataClient.java

This Java class contains code that can invoke the four operations provided by the EmployeeDataService hosted by the Tomcat 7 server.

Listing 5-7. EmployeeDataClient.java class

```
package com.bemach.ws.cxf.client;
/**
 * 2013 (C) BEM, Inc., Fairfax, Virginia
 *
 * Unless required by applicable law or agreed to in writing,
 * software distributed is distributed on an
 * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
 * KIND, either express or implied.
 *
 */

import java.net.MalformedURLException;
import java.net.URL;
import java.util.logging.Logger;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;

import com.bemach.ws.cxf.Employee;
import com.bemach.ws.cxf.EmployeeDataIf;
import com.bemach.ws.cxf.EmployeeDataService;
import com.bemach.ws.cxf.SOAPException_Exception;

/**
 * This class is a wrapper class for accessing a remote web service.
 *
 * @author ktran
 * @version 1.0
 */
public class EmployeeDataClient {
   private static final Logger LOG = Logger.getLogger(EmployeeDataClient.class.
getName());
   private EmployeeDataIf emplDs = null;

   /**
    * This method constructs the EmployeeDataClient object.
    *
    * @param urlStr - The URL of the Web Service
    * @param targetNs - Namespace of the target web service
    * @param name - the name of the web service.
    * @throws MalformedURLException - if an error occurs, exception is thrown.
    *
    */
   public EmployeeDataClient(String urlStr, String targetNs, String name)
         throws MalformedURLException {
      LOG.info("Constructor ...");
      QName qName = new QName(targetNs, name);
      URL url = new URL(urlStr);
      Service service = EmployeeDataService.create(url, qName);
      emplDs = service.getPort(EmployeeDataIf.class);
   }
```

```
    /**
     * This method gets an employee record based on a unique id.
     *
     * @param id - unique employee id.
     * @return - an employee record.
     * @throws SOAPException_Exception - an exception is thrown.
     */
    public Employee get(long id) throws SOAPException_Exception {
        return emplDs.getEmployee(id);
    }

    /**
     * This method creates an employee record.
     *
     * @param empl
     * @return
     */
    public long create(Employee empl) {
        return emplDs.createEmployee(empl);
    }

    /**
     * This method deletes an employee record based on a unique id.
     *
     * @param id
     * @return
     */
    public boolean delete(long id) {
        return emplDs.deleteEmployee(id);
    }

    /**
     * This method updates an employee record with a new one.
     *
     * @param empl
     * @return
     */
    public boolean update(Employee empl) {
        return emplDs.updateEmployee(empl);
    }

    /**
     * This is the main entrance of the code.
     *
     * @param args
     * @throws MalformedURLException
     * @throws SOAPException_Exception
     */
```

```
   public static void main(String[] args)
         throws MalformedURLException, SOAPException_Exception {
      LOG.info("Calling Employee (CXF) data service ... ");
      String targetNameSpace = "http://cxf.ws.bemach.com/";
      String name = "EmployeeDataService";
      String portNo = args[0];
      String urlStr = String.format("http://localhost:%s/cxf-ws/
employees?WSDL",portNo);

      EmployeeDataClient cli = new EmployeeDataClient(urlStr, targetNameSpace,
name);
      LOG.info("URL="+urlStr);

      long oldEmplNo = Integer.valueOf(args[1]);
      Employee empl = cli.get(oldEmplNo);
      LOG.info("last="+empl.getLastName());
      LOG.info("hire="+empl.getHireDate());
      LOG.info("last="+empl.getLastName());
      LOG.info("first="+empl.getFirstName());

      empl.setFirstName("Silvester");
      empl.setLastName("Johnny");
      long newEmplNo = cli.create(empl);
      LOG.info("emplNo="+newEmplNo);

      Employee newEmpl = cli.get(newEmplNo);

      newEmpl.setLastName("New-name");
      newEmpl.setEmplNo(newEmplNo);
      boolean status = cli.update(newEmpl);
      LOG.info("update:"+status);
      LOG.info("last="+newEmpl.getLastName());
      LOG.info("first="+newEmpl.getFirstName());

      status = cli.delete(newEmplNo);
      LOG.info("deleteEmployee:"+status);
      LOG.info("Exit!");
   }
}
```

In this class, we provide the port number as the first program argument. The Tomcat port number is 8080.

5.6.3 Build a Java library

This build script creates a Java library called 'cxf-ws-client.jar'. It requires the cxf-ws-generated.jar library that is stored in the dist directory.

Listing 5-8. Content of build.xml for cxf-ws-client Java Project

```
<project name="cxf-ws-client" default="dist" basedir=".">
   <description>
         Client for employees data service.
      Assumed that cxf-ws-generated.jar is generated and built elsehwere
   </description>

   <!- set global properties for this build ->
   <property environment="env"/>
   <path id="classpath.base">
      <fileset dir="./lib" includes="**/*.jar" />
   </path>

   <path id="classpath.compile">
      <path refid="classpath.base"/>
   </path>

   <target name="init">
      <mkdir dir="./bin"/>
      <mkdir dir="./dist"/>
   </target>

   <target name="compile" depends="init" description="compile the source " >
      <javac srcdir="./src" destdir="./bin" debug="true">
         <classpath refid="classpath.compile" />
      </javac>
   </target>

   <target name="dist" depends="compile" description="generate the distribution" >
      <!- Create the distribution directory ->
      <jar jarfile="./dist/cxf-ws-client.jar" basedir="./bin"/>

   </target>

   <target name="clean" description="clean up" >
      <!- Delete the ${build} directory trees ->
      <delete dir="./dist"/>
      <delete dir="./bin"/>
   </target>
</project>
```

5.6.4 Run the Client Application

To run the client application, go to the cxf-ws-client project directory. At the command prompt, run the following command:

```
java -cp ./lib/cxf-ws-generated.jar;./dist/cxf-ws-client.jar com.bemach.
ws.cxf.client.EmployeeDataClient 8080
```

The output is as follows:

```
Calling Employee (CXF) data service ...
URL=http://localhost:8080/cxf-ws/employees?WSDL
last=Fecello
hire=1986-06-26T00:00:00-04:00
last=Fecello
first=Silvester
emplNo=500001
update:true
last=New-name
first=Silvester
deleteEmployee:true
Exit!
```

6 Apache CXF and Oracle WebLogic Server

Objectives

After completing this chapter, you should be able to:

1. Understand the fundamentals of Oracle WebLogic 12
2. Create a basic WebLogic domain
3. Start an Administration Server
4. Use WebLogic to deploy a CXF WS application on the server
5. Test Web Service using a WebLogic Console

In this chapter, we take the same Web application – cxf-ws.war – and deploy it on a WebLogic Server version 12.

6.1 Oracle WebLogic Server 12

Oracle WebLogic Server (WLS) is also known as the 'Oracle Fusion Middleware'. The latest release of WLS is 12.1.1. Oracle WLS is an industrial-strength enterprise-ready Java platform, Enterprise Edition (Java EE) application server. It is the foundation for building Service-oriented Architectures (SOA) applications using Oracle software products.

Oracle WLS implements complete JEE 6 specification and provide a set of APIs for creating a variety of services: databases, messaging, and connections to external systems. Oracle WLS provides an environment capable of deploying mission-critical applications that are robust, secure, highly available and scalable.

Major advantages of using Oracle WLS are briefly described in the following sections:

6.1.1 Programming Modles

Oracle WLS comes with a set of tools that enable the following capabilities:

- Web Applications (JSP and Servlet)
- Web Services (JAX-WS, JAX-RPC, JAX-RS)
- XML Programming (JAXB)
- Java Messaging Service (JMS)
- Java Database Connectivity (JDBC)
- Resource Adapters (Enterprise Information Systems)

- Enterprise JavaBeans (EJB)
- Remote Method Invocation (RMI)
- Security APIs (Security Service Providers APIs)
- WebLogic Tuxedo Connectivity (WTC)

6.1.2 Highly Availability

Mission-critical applications can be supported with the following capabilities:

- WebLogic Server Clusters
- Work Managers
- Overload Protection
- Network Channels
- Weblogic Server Peristent Store
- Store-and-forward Services
- Enterprise-ready Development Tools
- Production Redeployment

6.2 Deployment Diagram

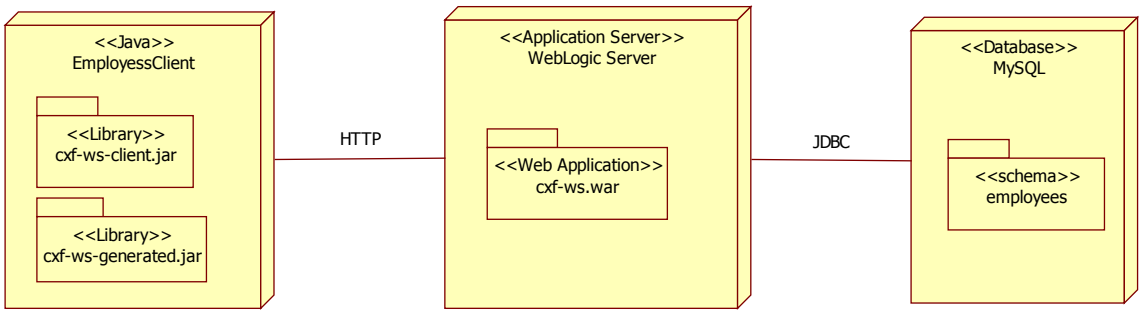


Figure 6-1. *Deployment Diagram for CXF Web Service Application and Oracle WebLogic Server*

6.3 Creating a WebLogic Domain

A WebLogic Server administration domain is a logical group of WLS resources. A domain is managed by a special type of server called an 'Administration Server'. This server instance is used for managing resources and configurations of these resources. Applications and services should not be deployed in an Administration Server; they should be deployed on Managed Server instances instead. A WLS domain may have one or more Managed Server instances.

Two or more Managed Servers can be grouped into a cluster. A domain can administer one or more clusters. For the sake of simplicity, we will deploy the CXF Web Service Application on an Administration Server. First, we create a WLS domain by following these steps:

1. Assuming that you installed Oracle WebLogic Server on Windows, go to C:\Oracle\Middleware\wlserver_12.1\common\bin.
2. Run config.cmd (or config.sh).
3. Follow the on-screen instructions to complete the creation of the domain.
4. Choose all default parameters.
5. Once complete, the domain is created and stored here: C:\Oracle\Middleware\user_projects\domains\base_domain

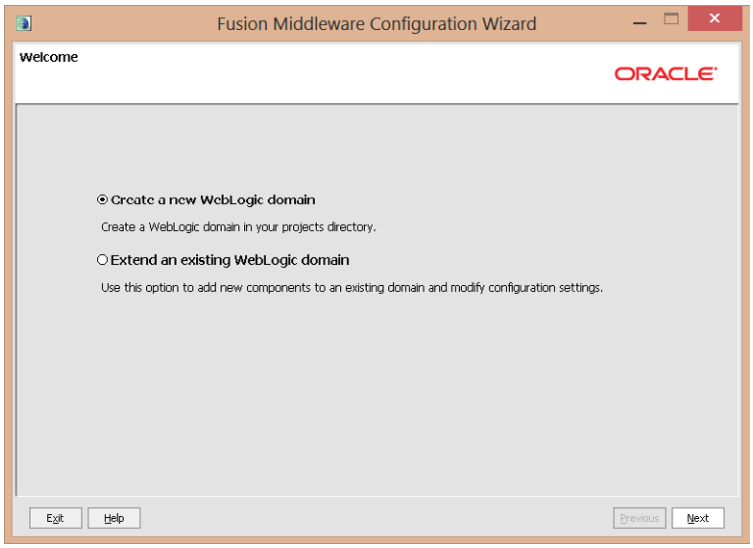


Figure 6-2. *Creating a WLS Domain*

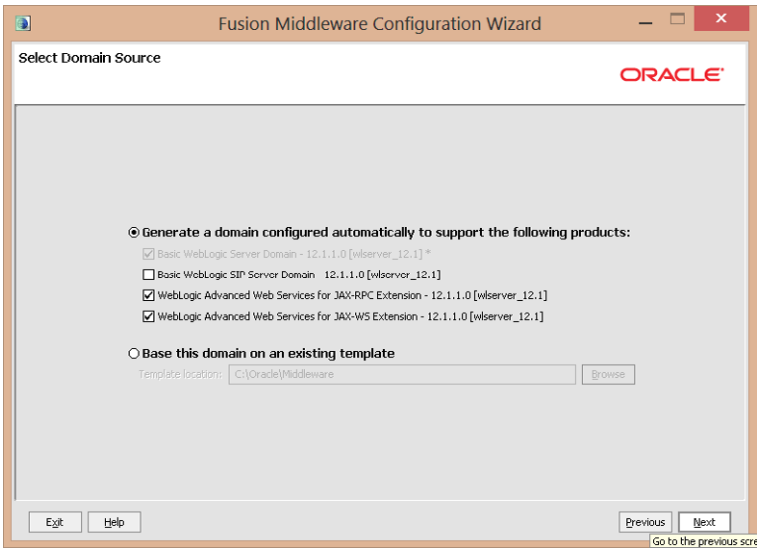


Figure 6-3. *Adding Extensions (JAX-WS and JAX-RPC)*

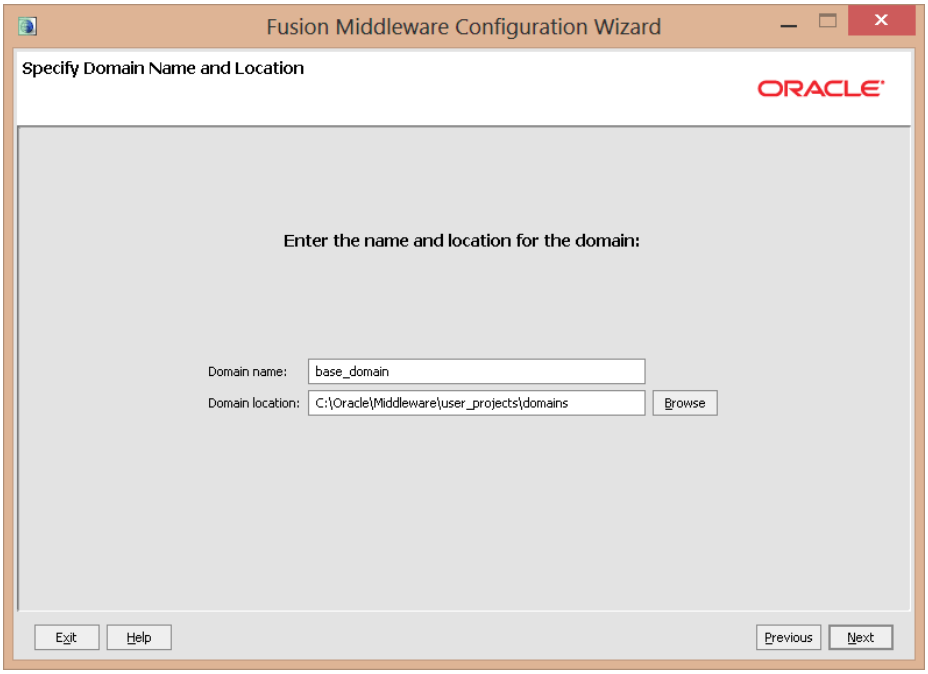


Figure 6-4. *Enter the Domain Name*

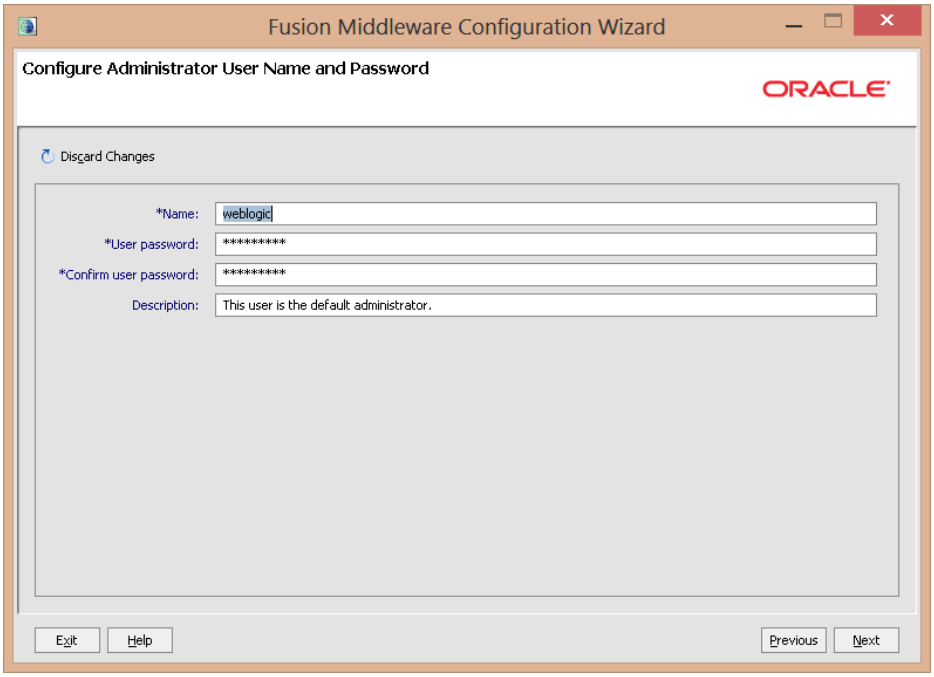


Figure 6-5. *Enter User ID and Password*

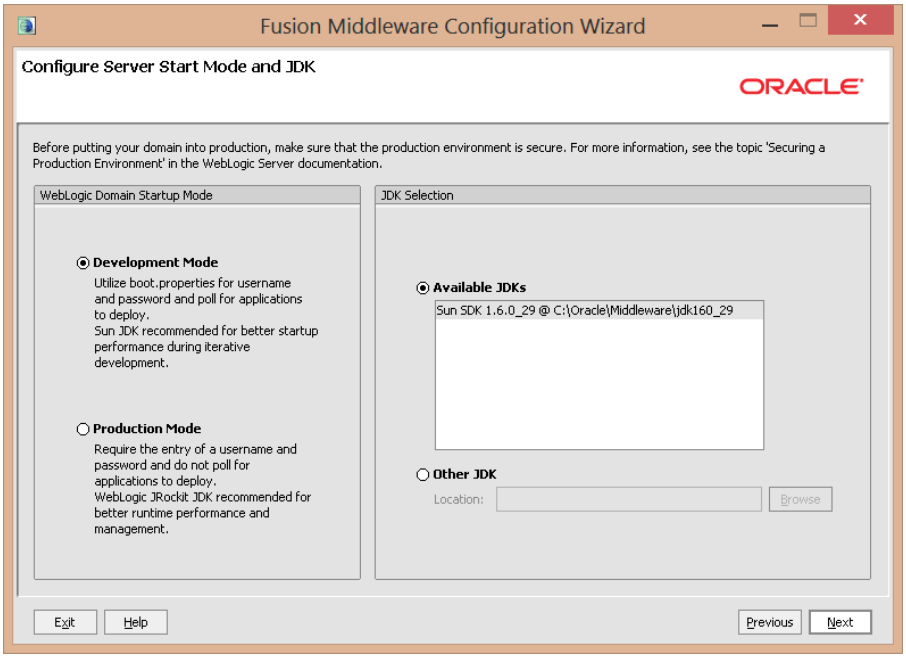


Figure 6-6. *Select a default JDK*

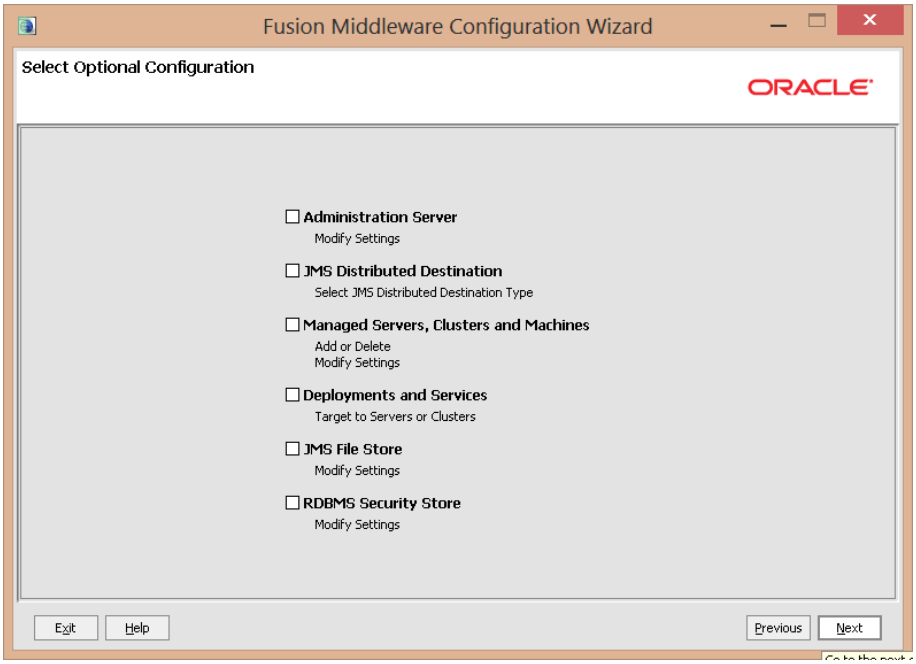


Figure 6-7. *Additional Configuration*

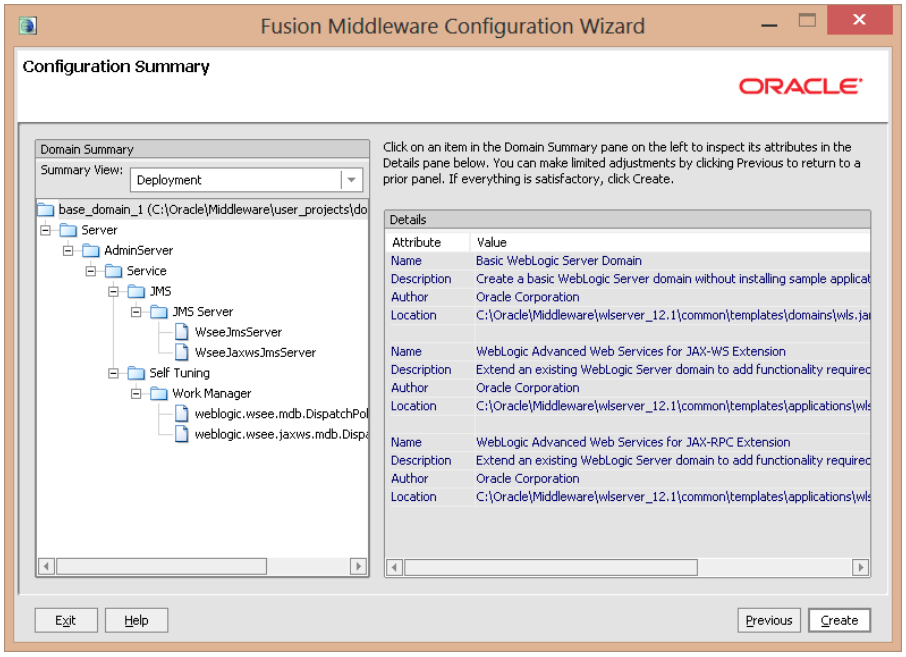


Figure 6-8. *Configuration Summary of the Domain*

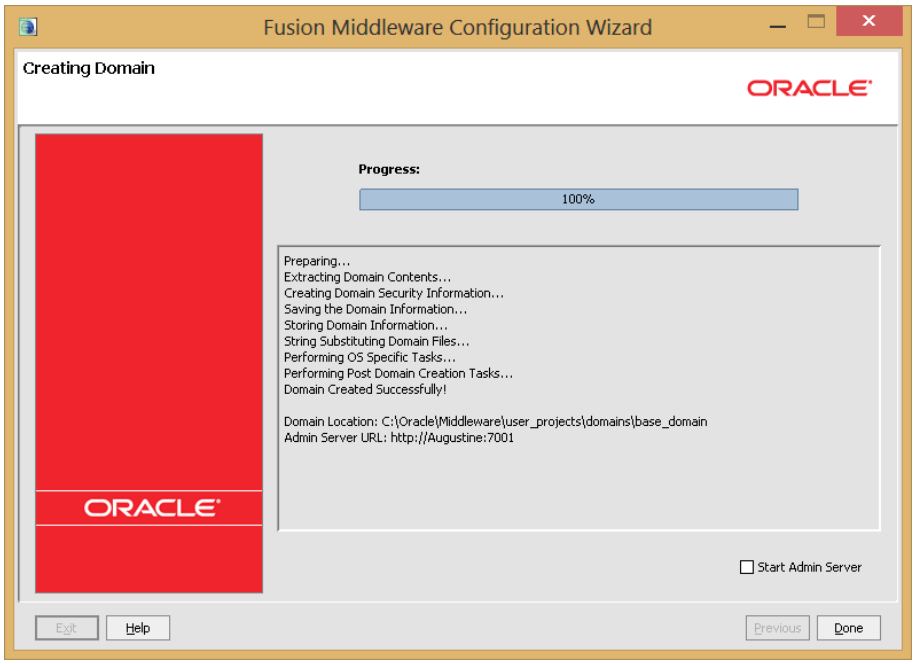


Figure 6-9. *Status of the Domain Creation*

6.3.1 Starting an Administration Server

To start an Administration Server:

1. Go to the domain directory:

   ```
   C:\Oracle\Middleware\user_projects\domains\base_domain
   ```

2. Run the following command:

   ```
   startweblogic.cmd
   ```

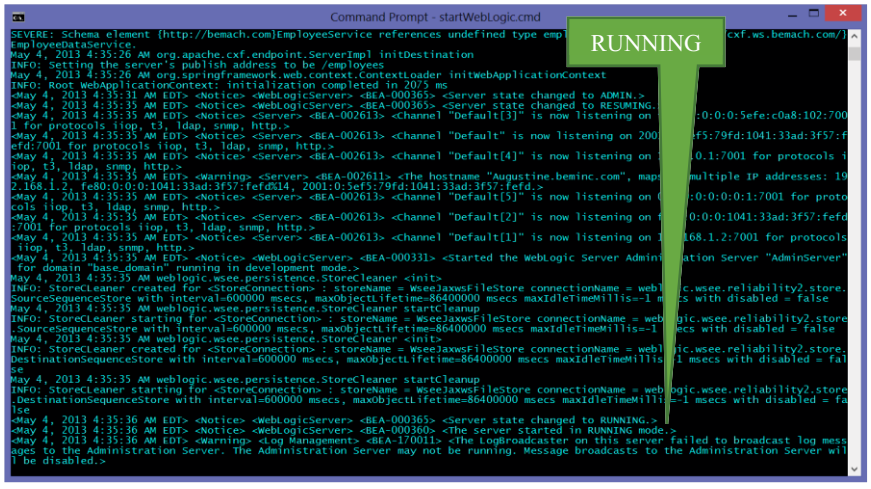


Figure 6-10. *Output of a WLS Administration Server*

Once the server has started successfully, it displays the following message in the command window:

```
<The server started in RUNNING mode.>
```

6.4 Deploy the Web Service

Make sure to include the weblogic.xml file in the WEB-INF directory of the cxf-ws project prior to building the Java Web Application.

6.4.1 weblogic.xml

This file contains WebLogic-specific configuration parameters. It is needed for deploying the CXF WS application on a WebLogic server.

Listing 6-1. Content of weblogic.xml to be included for cxf-ws.war Web Application

```
<?xml version="1.0" encoding="UTF-8"?>
<weblogic-web-app xmlns="http://xmlns.oracle.com/weblogic/weblogic-web-app"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.oracle.com/weblogic/weblogic-web-app
http://xmlns.oracle.com/weblogic/weblogic-web-app/1.0/weblogic-web-app.xsd">
      <context-root>cxf-ws</context-root>
      <jsp-descriptor>
            <precompile>true</precompile>
      </jsp-descriptor>

      <session-descriptor>
            <timeout-secs>900</timeout-secs>
            <invalidation-interval-secs>10</invalidation-interval-secs>
            <max-in-memory-sessions>500</max-in-memory-sessions>
      </session-descriptor>

</weblogic-web-app>
```

To deploy the CXF Web Application on a WebLogic Server, take the following steps:

1. Open a browser and go to http://localhost:7001/console. Login as weblogic/weblogic1 using the username/password that you defined during the domain creation step.

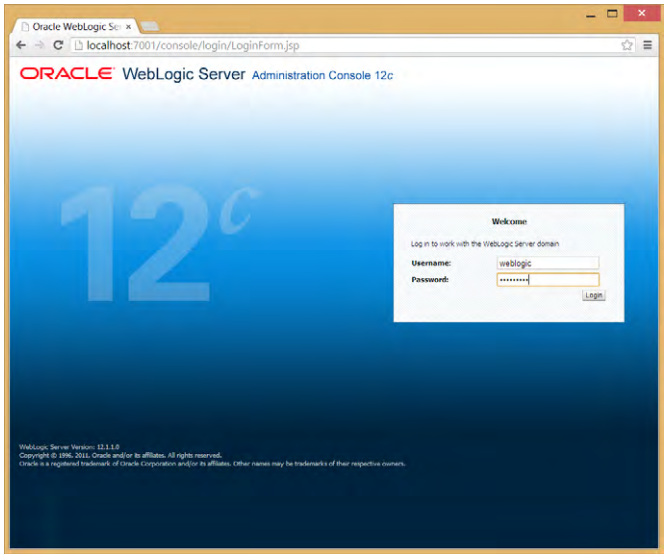


Figure 6-11. *OracleWLS Console Login Screen*

2. On the left panel, choose the `Deployments option`.

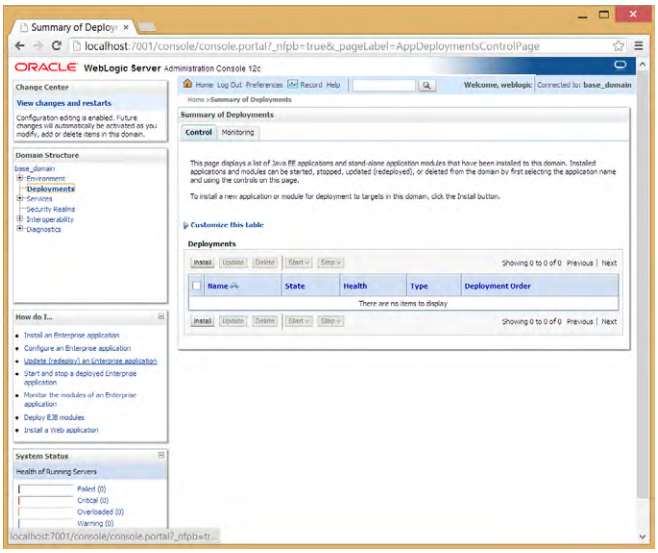


Figure 6-12. *Oracle WLS Deployment Screen*

3. On the right panel, click the Install button.
4. Choose the cxf-ws.war file located in the dist directory of the cxf-ws project.

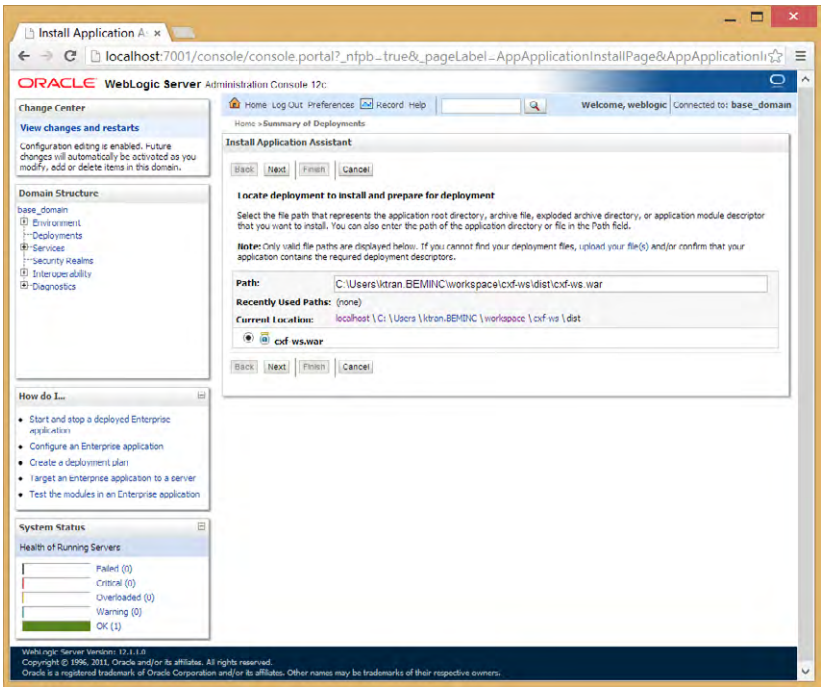


Figure 6-13. *Oracle WLS Install Application Screen*

5. Click 'Next'.

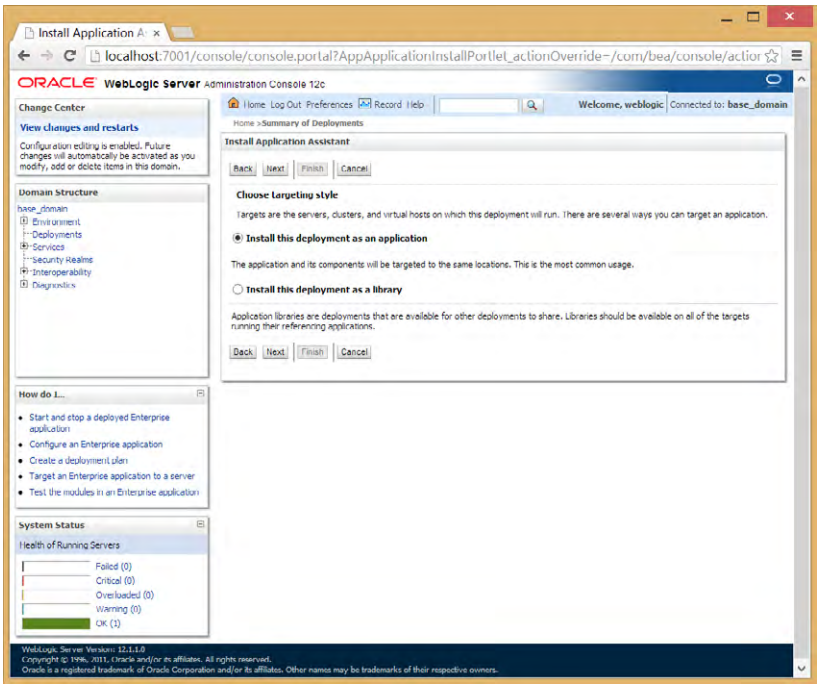


Figure 6-14. *Type of Deployment*

6. Click 'Next'.

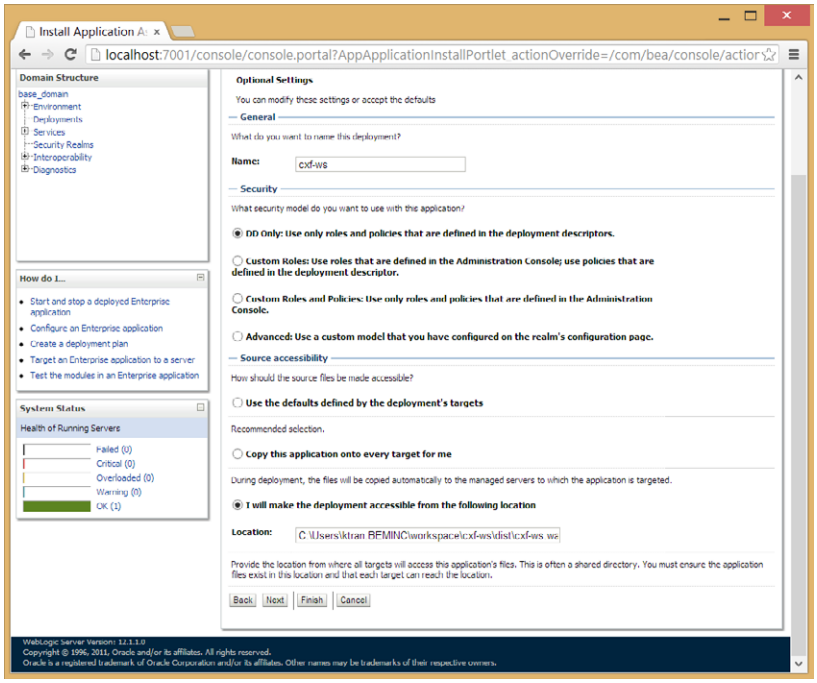


Figure 6-15. *Additional Settings for the Deployment Application Process*

7. Click 'Finish'.

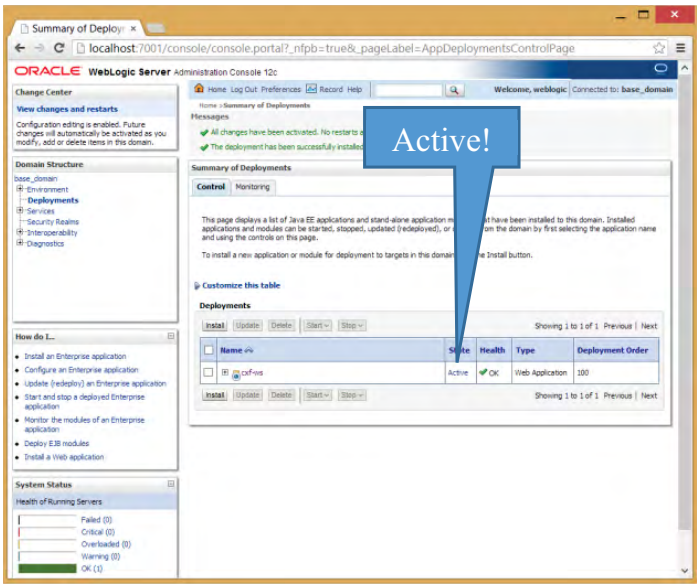


Figure 6-16. *Deployment Verification*

The state must show as 'Active'. Any other state can be a problem.

We are now ready to test the CXF Web Service that is hosted by an Oracle WebLogic Server.

6.5 Test CXF Web Service with WebLogic Test Tools

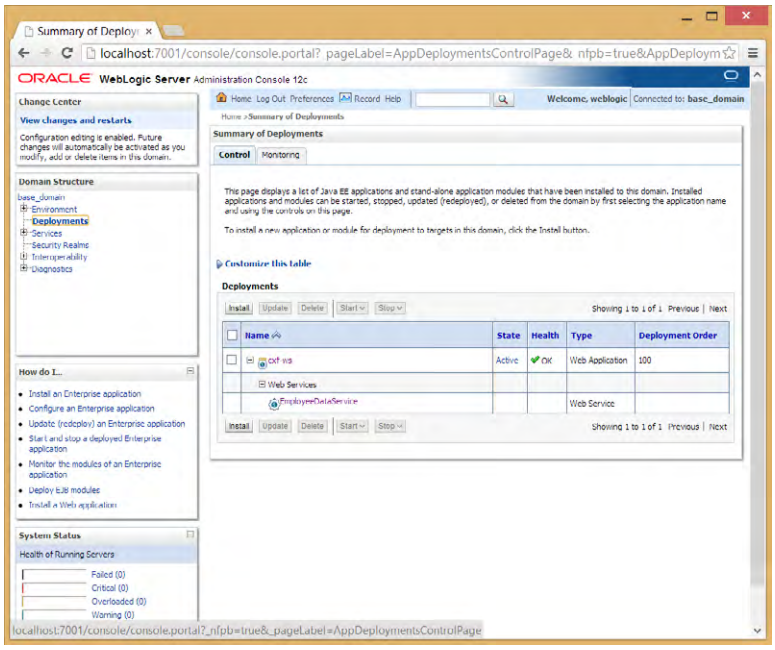


Figure 6-17. *Select the Web Service Application for Testing*

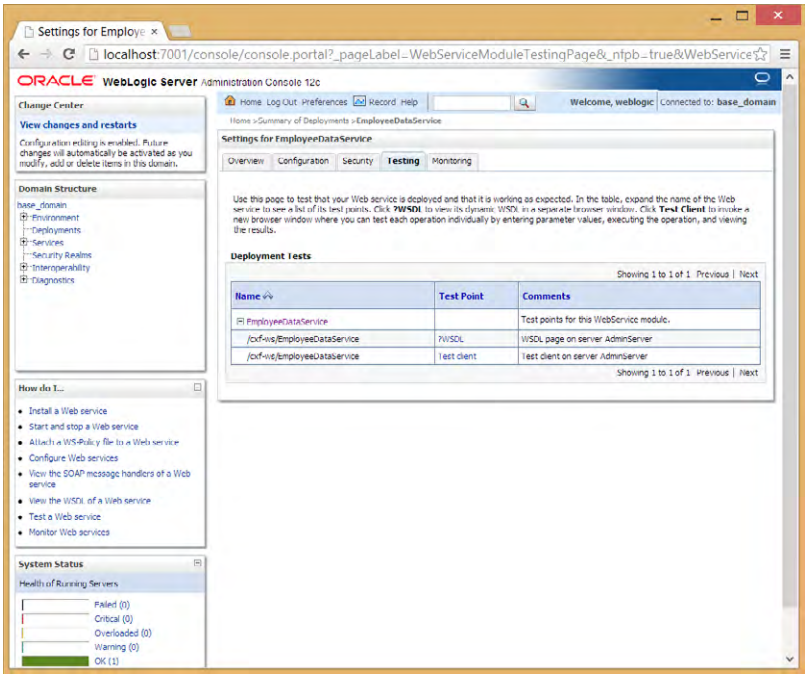


Figure 6-18. *Display of the Web Application*

Click on *Test Client URL*.

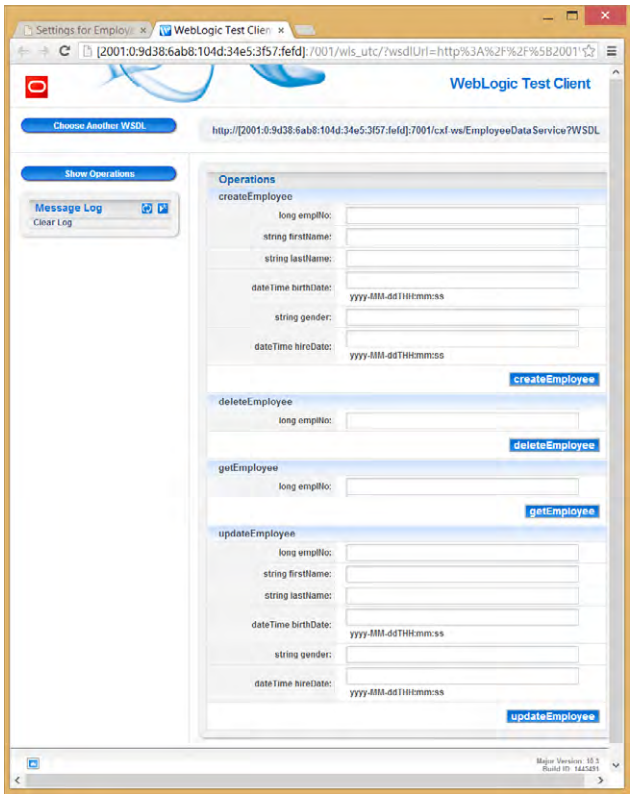


Figure 6-19. *WebLogic Test Client*

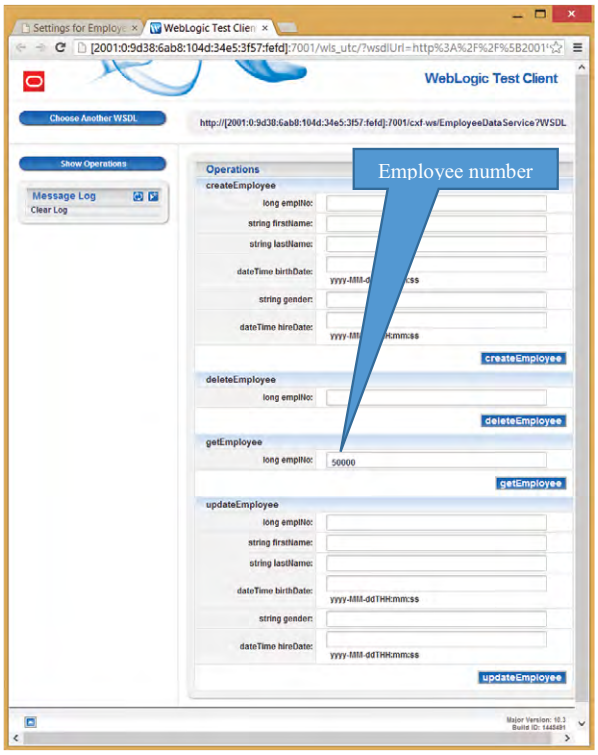


Figure 6-20. *Prepare to Run getEmployee Operation*

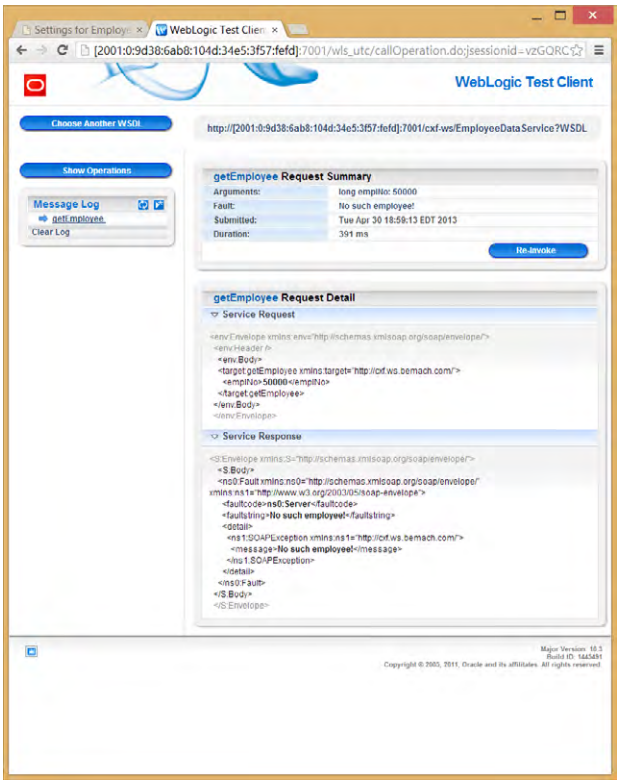


Figure 6-21. *Result of a call to getEmployee Operation*

6.5.1 Check WSDL

The WSDL is located at the following URL:

http://localhost:7001/cxf-ws/employees?wsdl

Listing 6-2. WSDL for CXF Web Application on Oracle WebLogic Server

```
<wsdl:definitions xmlns:ns1="http://schemas.xmlsoap.org/soap/http"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://cxf.ws.bemach.com/"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   name="EmployeeDataService" targetNamespace="http://cxf.ws.bemach.com/">
   <wsdl:types>
      <xsd:schema xmlns="http://cxf.ws.bemach.com/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
         attributeFormDefault="unqualified" elementFormDefault="unqualified"
         targetNamespace="http://cxf.ws.bemach.com/">
         <xsd:complexType name="employee">
            <xsd:sequence>
               <xsd:element name="emplNo" type="xsd:long" />
               <xsd:element minOccurs="0" name="firstName" type="xsd:string" />
               <xsd:element minOccurs="0" name="lastName" type="xsd:string" />
               <xsd:element minOccurs="0" name="birthDate" type="xsd:dateTime" />
               <xsd:element minOccurs="0" name="gender" type="xsd:string" />
               <xsd:element minOccurs="0" name="hireDate" type="xsd:dateTime" />
            </xsd:sequence>
         </xsd:complexType>
         <xsd:complexType name="SOAPException">
            <xsd:sequence />
         </xsd:complexType>
         <xsd:element name="SOAPException" type="SOAPException" />
      </xsd:schema>
      <xsd:schema xmlns:ns0="http://bemach.com"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
         attributeFormDefault="unqualified" elementFormDefault="unqualified"
         targetNamespace="http://bemach.com">
         <xsd:import namespace="http://cxf.ws.bemach.com/" />
         <xsd:element name="EmployeeService" type="employee" />
      </xsd:schema>
   </wsdl:types>
   <wsdl:message name="createEmployeeResponse">
      <wsdl:part name="return" type="xsd:long"></wsdl:part>
   </wsdl:message>
   <wsdl:message name="getEmployeeResponse">
      <wsdl:part name="return" type="tns:employee"></wsdl:part>
   </wsdl:message>
   <wsdl:message name="updateEmployee">
      <wsdl:part name="employee" type="tns:employee"></wsdl:part>
   </wsdl:message>
   <wsdl:message name="SOAPException">
      <wsdl:part element="tns:SOAPException" name="SOAPException"></wsdl:part>
   </wsdl:message>
```

```
<wsdl:message name="updateEmployeeResponse">
      <wsdl:part name="return" type="xsd:boolean"></wsdl:part>
   </wsdl:message>
   <wsdl:message name="deleteEmployeeResponse">
      <wsdl:part name="return" type="xsd:boolean"></wsdl:part>
   </wsdl:message>
   <wsdl:message name="getEmployee">
      <wsdl:part name="emplNo" type="xsd:long"></wsdl:part>
   </wsdl:message>
   <wsdl:message name="createEmployee">
      <wsdl:part name="employee" type="tns:employee"></wsdl:part>
   </wsdl:message>
   <wsdl:message name="deleteEmployee">
      <wsdl:part name="emplNo" type="xsd:long"></wsdl:part>
   </wsdl:message>
   <wsdl:portType name="EmployeeDataIf">
      <wsdl:operation name="createEmployee">
         <wsdl:input message="tns:createEmployee"
name="createEmployee"></wsdl:input>
         <wsdl:output message="tns:createEmployeeResponse"
name="createEmployeeResponse"></wsdl:output>
      </wsdl:operation>
      <wsdl:operation name="deleteEmployee">
         <wsdl:input message="tns:deleteEmployee"
name="deleteEmployee"></wsdl:input>
         <wsdl:output message="tns:deleteEmployeeResponse"
name="deleteEmployeeResponse"></wsdl:output>
      </wsdl:operation>
      <wsdl:operation name="updateEmployee">
         <wsdl:input message="tns:updateEmployee"
name="updateEmployee"></wsdl:input>
         <wsdl:output message="tns:updateEmployeeResponse"
name="updateEmployeeResponse"></wsdl:output>
      </wsdl:operation>
      <wsdl:operation name="getEmployee">
         <wsdl:input message="tns:getEmployee"
name="getEmployee"></wsdl:input>
         <wsdl:output message="tns:getEmployeeResponse"
name="getEmployeeResponse"></wsdl:output>
         <wsdl:fault message="tns:SOAPException" name="SOAPException"></wsdl:fault>
      </wsdl:operation>
   </wsdl:portType>
   <wsdl:binding name="EmployeeDataServiceSoapBinding" type="tns:EmployeeDataIf">
      <soap:binding style="rpc"
         transport="http://schemas.xmlsoap.org/soap/http" />
      <wsdl:operation name="createEmployee">
         <soap:operation soapAction="" style="rpc" />
         <wsdl:input name="createEmployee">
            <soap:body namespace="http://cxf.ws.bemach.com/" use="literal" />
         </wsdl:input>
```

```
         <wsdl:output name="createEmployeeResponse">
            <soap:body namespace="http://cxf.ws.bemach.com/" use="literal" />
         </wsdl:output>
      </wsdl:operation>
      <wsdl:operation name="deleteEmployee">
         <soap:operation soapAction="" style="rpc" />
         <wsdl:input name="deleteEmployee">
            <soap:body namespace="http://cxf.ws.bemach.com/" use="literal" />
         </wsdl:input>
         <wsdl:output name="deleteEmployeeResponse">
            <soap:body namespace="http://cxf.ws.bemach.com/" use="literal" />
         </wsdl:output>
      </wsdl:operation>
      <wsdl:operation name="getEmployee">
         <soap:operation soapAction="" style="rpc" />
         <wsdl:input name="getEmployee">
            <soap:body namespace="http://cxf.ws.bemach.com/" use="literal" />
         </wsdl:input>
         <wsdl:output name="getEmployeeResponse">
            <soap:body namespace="http://cxf.ws.bemach.com/" use="literal" />
         </wsdl:output>
         <wsdl:fault name="SOAPException">
            <soap:fault name="SOAPException" use="literal" />
         </wsdl:fault>
      </wsdl:operation>
      <wsdl:operation name="updateEmployee">
         <soap:operation soapAction="" style="rpc" />
         <wsdl:input name="updateEmployee">
            <soap:body namespace="http://cxf.ws.bemach.com/" use="literal" />
         </wsdl:input>
         <wsdl:output name="updateEmployeeResponse">
            <soap:body namespace="http://cxf.ws.bemach.com/" use="literal" />
         </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>
   <wsdl:service name="EmployeeDataService">
      <wsdl:port binding="tns:EmployeeDataServiceSoapBinding"
         name="EmployeeDataPort">
         <soap:address location="http://localhost:7001/cxf-ws/employees" />
      </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```

6.6 Run the Client Application

```
java -cp ./lib/cxf-ws-generated.jar;./dist/cxf-ws-client.jar com.
bemach.ws.cxf.client.EmployeeDataClient 7001
```

The output of this test will be printed on-screen as follows:

```
EmployeeDataClient 7001
Calling Employee (CXF) data service ...
URL=http://localhost:7001/cxf-ws/employees?WSDL
last=Fecello
hire=1986-06-26T00:00:00.0-04:00
last=Fecello
first=Silvester
emplNo=500001
update:true
last=New-name
first=Silvester
deleteEmployee:true
Exit!
```

7 Appendix A – Development Environment

Many development tools are available today for Java developers to utilize when developing Web Services. Two in particular are recommended and used throughout this book.

7.1 Install Java Development Kit (JDK) 6

Visit Oracle's website (http://www.oracle.com) and download Java Platform (JDK) 6. Follow the installation instructions and use all defaults.

JDK 7 can be used to run all of the examples in this book; however, it would be difficult to work with Oracle WebLogic at this time because this software package still runs with JDK 6. Oracle WebLogic 12 can be made to work with JDK 7, but it takes more time to set up.

7.2 Install Eclipse Interactive Development Environment (IDE)

Visit Eclipse's website (http://www.eclipse.org) and download the latest version of Eclipse IDE for Java EE developers.

After installing Eclipse on your system, run it by double-clicking on the Eclipse icon in the installed folder. The two basic project types that are used in this book are Java Projects and Dynamic Web Projects. Instructions for creating these projects are provided in the following sections.

7.2.1 Create a Java project in Eclipse

To create a Java project, take the following steps:

1. Select File (menu) → New → Java Project. A Project creation screen will pop up. Enter your project name and click 'Next' at the bottom.

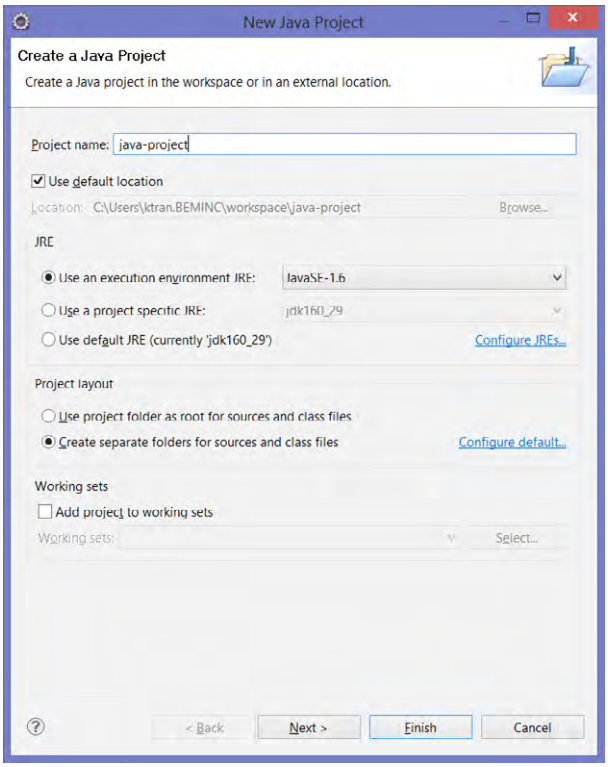


Figure 7-1. *Create a Java Project for the Eclipse IDE*

2. On the Java Settings screen, choose all default values and click 'Finish' at the bottom.

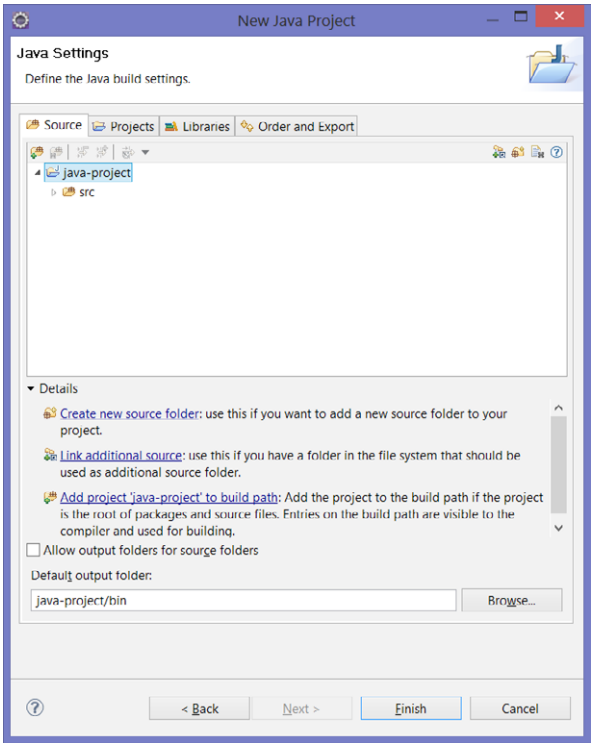


Figure 7-2. *Java Settings Screen*

7.2.2 Create a Dynamic Web project in Eclipse

1. Choose File (menu) → New → Other… A 'Select a wizard' window pops up as shown below:

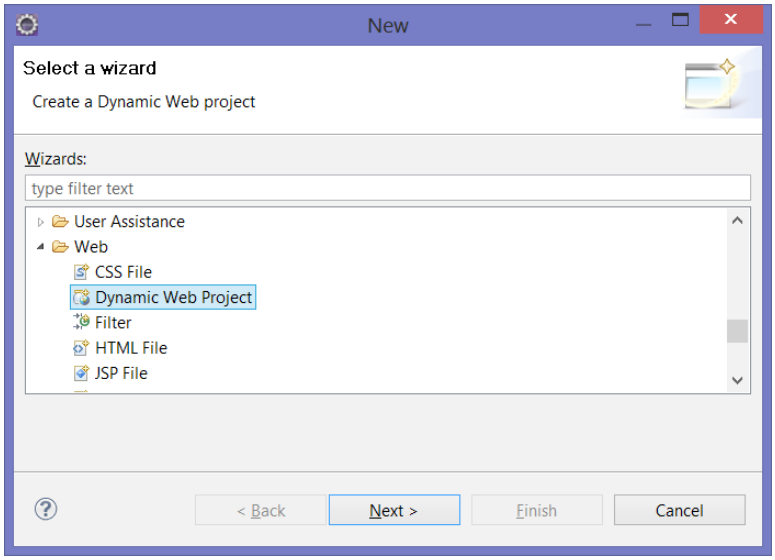


Figure 7-3. *Create a Dynamic Web Project*

2. Enter the project name and click 'Next' at the bottom of the screen.

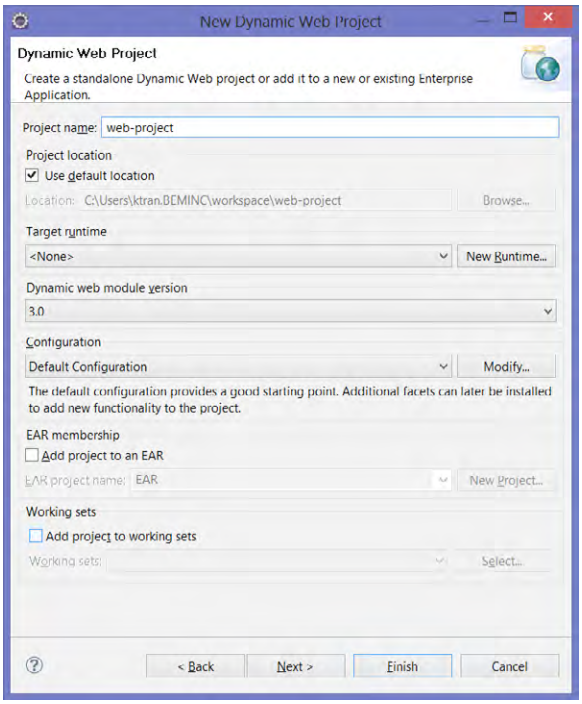


Figure 7-4. *Options for a Dynamic Web Project*

3. Click 'Next' on the Java screen.

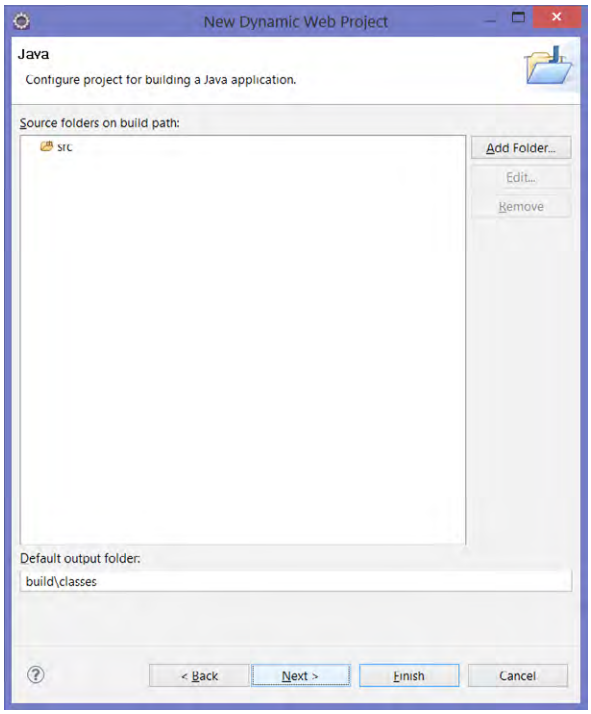


Figure 7-5. *Additional Java Options*

4. Make sure to check 'Generate web.xml deployment descriptor'.

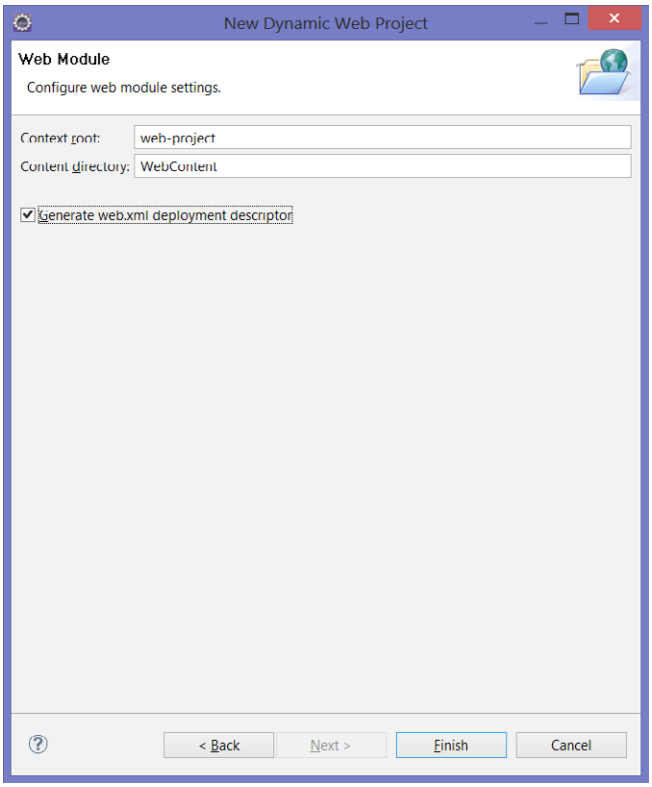


Figure 7-6. *Finishing the Creation of a Dynamic Web Project*

7.3 Install MySQL Community Server Database

Visit MySQL's website (http://www.mysql.com) and download the MySQL Community Server. This version is freely available to developers.

Run the installer and choose the default options. Make sure to remember your root password.

Download and install Employees, a sample database, from MySQL's website: http://dev.mysql.com/ doc/ employee/en/index.html. Follow the installation instructions and use all default options.

Create a user called 'empl_1' and assign the Employees database.

If you'd prefer a simpler approach, create an Employees database with just a single table 'Employees' as shown below.

Listing 7-1. A DDL for creating employees table

```
CREATE TABLE employees (
   emp_no       INT             NOT NULL,
   birth_date   DATE            NOT NULL,
   first_name   VARCHAR(14)     NOT NULL,
   last_name    VARCHAR(16)     NOT NULL,
   gender       ENUM ('M','F')  NOT NULL,
   hire_date    DATE            NOT NULL,
   PRIMARY KEY (emp_no)
);
```

Download and extract the JDBC driver for accessing MySQL database:
http://dev.mysql.com/downloads/connector/j/

7.4 Install Oracle Fusion Middleware Software

Visit Oracle's website (http://www.oracle.com) and download Oracle Fusion Middleware (formerly 'WebLogic Server') version 12.1.1. Oracle may require the user to sign up for an account. This version is fully functional for developers; however, it is limited to five (5) client connections. Make sure to download the Windows or Linux Installer version. Warning: this is a large download (1.2 GB). The installer version includes a JDK and a customized Eclipse IDE for Oracle's middleware. Run the installer and choose all default options.

7.5 Install Apache Tomcat server

Visit Apache Tomcat's website (http://tomcat.apache.org) and download the latest Tomcat version 7. Tomcat is an open source JEE application server that is available for free download.

Verify the Tomcat server by visiting http://localhost:8080 on a browser. The following page should appear:

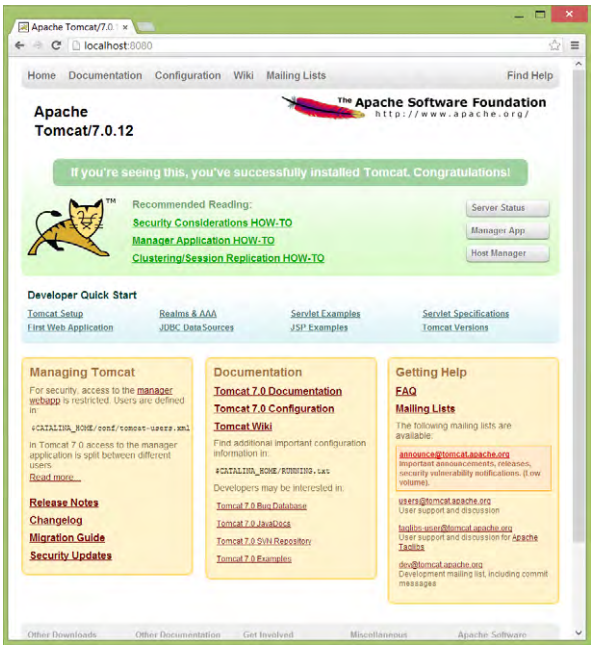


Figure 7-7. *Working Tomcat Console Screen*

7.6 Apache CXF

Visit the Apache CXF website to download the software package. Unpack the download into a directory. All necessary Java libraries are stored in the unpacked directory. Make sure to include appropriate library files (JAR) in the library of the cxf-ws Java project.

```
aopalliance-1.0.jar
asm-3.3.jar
axis.jar
commons-dbcp-1.4.jar
commons-discovery-0.2.jar
commons-logging-1.1.1.jar
commons-logging.jar
cxf-2.4.0.jar
geronimo-activation_1.1_spec-1.1.jar
geronimo-annotation_1.0_spec-1.1.1.jar
geronimo-javamail_1.4_spec-1.7.1.jar
```

```
geronimo-servlet_3.0_spec-1.0.jar
geronimo-ws-metadata_2.0_spec-1.1.3.jar
jaxb-api-2.2.1.jar
jaxb-impl-2.2.1.1.jar
jaxrpc.jar
neethi-3.0.0.jar
saaj-api-1.3.jar
saaj-impl-1.3.2.jar
spring-aop-3.0.5.RELEASE.jar
spring-asm-3.0.5.RELEASE.jar
spring-beans-3.0.5.RELEASE.jar
spring-context-3.0.5.RELEASE.jar
spring-core-3.0.5.RELEASE.jar
spring-expression-3.0.5.RELEASE.jar
spring-web-3.0.5.RELEASE.jar
stax2-api-3.1.1.jar
woodstox-core-asl-4.1.1.jar
wsdl4j-1.6.2.jar
wsdl4j.jar
xml-resolver-1.2.jar
xmlschema-core-2.0.jar
```

7.7 Install SOAPUI software

Visit SOAPUI's website (http://www.soapui.org) and download a free version of SOAPUI software.

7.8 Source Code

All Java code for this book can be found at Dr. Tran's blog: http://drtran.bemach.com.

8 Endnotes

1. Note that '**employees**' is the actual name of the database.
2. Note that 'employees' is the name of a table of 'employees' database.